W3School JavaScript & jQuery 教程

wizardforcel

Published with GitBook



目錄

W3School Javascript & jQuery 教程

来源：

- JavaScript教程
- HTML DOM教程
- JavaScript高级教程
- jQuery教程
- Ajax教程
- Google地图教程

整理：飞龙

JavaScript 基础

JavaScript 简介

JavaScript 是世界上最流行的编程语言。

这门语言可用于 **HTML** 和 **web**，更可广泛用于服务器、**PC**、笔记本电脑、平板电脑和智能手机等设备。

JavaScript 是脚本语言

JavaScript 是一种轻量级的编程语言。

JavaScript 是可插入 HTML 页面的编程代码。

JavaScript 插入 HTML 页面后，可由所有的现代浏览器执行。

JavaScript 很容易学习。

您将学到什么

下面是您将在本教程中学到的主要内容。

JavaScript：写入 HTML 输出

实例

```
document.write("<h1>This is a heading</h1>");
document.write("<p>This is a paragraph</p>");
```

提示：您只能在 HTML 输出中使用 document.write。如果您在文档加载后使用该方法，会覆盖整个文档。

JavaScript：对事件作出反应

实例

```
<button type="button" onclick="alert('Welcome!')">点击这里</button>
```

alert() 函数在 JavaScript 中并不常用，但它对于代码测试非常方便。

onclick 事件只是您即将在本教程中学到的众多事件之一。

JavaScript：改变 HTML 内容

使用 JavaScript 来处理 HTML 内容是非常强大的功能。

实例

```
x=document.getElementById("demo")  //查找元素
x.innerHTML="Hello JavaScript";    //改变内容
```

您会经常看到 document.getElementByID("*some id*")。这个方法是 HTML DOM 中定义的。

DOM（文档对象模型）是用以访问 HTML 元素的正式 W3C 标准。

您将在本教程的多个章节中学到有关 HTML DOM 的知识。

JavaScript：改变 HTML 图像

本例会动态地改变 HTML <image> 的来源 (src)：

The Light bulb

点击灯泡就可以打开或关闭这盏灯

JavaScript 能够改变任意 HTML 元素的大多数属性，而不仅仅是图片。

JavaScript：改变 HTML 样式

改变 HTML 元素的样式，属于改变 HTML 属性的变种。

实例

```
x=document.getElementById("demo")  //找到元素
x.style.color="#ff0000";           //改变样式
```

JavaScript：验证输入

JavaScript 常用于验证用户的输入。

实例

```
if isNaN(x) {alert("Not Numeric")};
```

您知道吗？

提示：JavaScript 与 Java 是两种完全不同的语言，无论在概念还是设计上。

Java（由 Sun 发明）是更复杂的编程语言。

ECMA-262 是 JavaScript 标准的官方名称。

JavaScript 由 Brendan Eich 发明。它于 1995 年出现在 Netscape 中（该浏览器已停止更新），并于 1997 年被 ECMA（一个标准协会）采纳。

课外阅读

JavaScript 高级教程： JavaScript 历史 、 JavaScript 实现

JavaScript 使用

HTML 中的脚本必须位于 **<script>** 与 **</script>** 标签之间。

脚本可被放置在 **HTML** 页面的 **<body>** 和 **<head>** 部分中。

JavaScript 输出

JavaScript 通常用于操作 **HTML** 元素。

操作 HTML 元素

如需从 JavaScript 访问某个 HTML 元素，您可以使用 document.getElementById(*id*) 方法。

请使用 "id" 属性来标识 HTML 元素：

例子

通过指定的 id 来访问 HTML 元素，并改变其内容：

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>

<p id="demo">My First Paragraph</p>

<script>
`document.getElementById("demo").innerHTML="My First JavaScript";`
</script>

</body>
</html>
```

JavaScript 由 web 浏览器来执行。在这种情况下，浏览器将访问 id="demo" 的 HTML 元素，并把它的内容（innerHTML）替换为 "My First JavaScript"。

写到文档输出

下面的例子直接把 <p> 元素写到 HTML 文档输出中：

实例

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>

<script>
`document.write("<p>My First JavaScript</p>");`
</script>

</body>
</html>
```

警告

请使用 document.write() 仅仅向文档输出写内容。

如果在文档已完成加载后执行 document.write，整个 HTML 页面将被覆盖：

实例

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>

<p>My First Paragraph.</p>

<button onclick="myFunction()">点击这里</button>

<script>
function myFunction()
{
document.write("糟糕！文档消失了。");
}
</script>

</body>
</html>
```

Windows 8 中的 JavaScript

提示：微软支持通过 JavaScript 创建 Windows 8 app。

对于因特网和视窗操作系统，JavaScript 都意味着未来。

JavaScript 语句

JavaScript 语句

JavaScript 语句向浏览器发出的命令。语句的作用是告诉浏览器该做什么。

下面的 JavaScript 语句向 id="demo" 的 HTML 元素输出文本 "Hello World"：

```
document.getElementById("demo").innerHTML="Hello World";
```

分号 ;

分号用于分隔 JavaScript 语句。

通常我们在每条可执行的语句结尾添加分号。

使用分号的另一用处是在一行中编写多条语句。

提示：您也可能看到不带有分号的案例。

在 JavaScript 中，用分号来结束语句是可选的。

JavaScript 代码

JavaScript 代码（或者只有 JavaScript）是 JavaScript 语句的序列。

浏览器会按照编写顺序来执行每条语句。

本例将操作两个 HTML 元素：

实例

```
document.getElementById("demo").innerHTML="Hello World";
document.getElementById("myDIV").innerHTML="How are you?";
```

JavaScript 代码块

JavaScript 语句通过代码块的形式进行组合。

块由左花括号开始，由右花括号结束。

块的作用是使语句序列一起执行。

JavaScript 函数是将语句组合在块中的典型例子。

下面的例子将运行可操作两个 HTML 元素的函数：

实例

```
function myFunction()
{
document.getElementById("demo").innerHTML="Hello World";
document.getElementById("myDIV").innerHTML="How are you?";
}
```

您将在稍后的章节学到更多有关函数的知识。

JavaScript 对大小写敏感。

JavaScript 对大小写是敏感的。

当编写 JavaScript 语句时，请留意是否关闭大小写切换键。

函数 getElementById 与 getElementbyID 是不同的。

同样，变量 myVariable 与 MyVariable 也是不同的。

空格

JavaScript 会忽略多余的空格。您可以向脚本添加空格，来提高其可读性。下面的两行代码是等效的：

```
var name="Hello";
var name = "Hello";
```

对代码行进行折行

您可以在文本字符串中使用反斜杠对代码行进行换行。下面的例子会正确地显示：

```
document.write("Hello \
World!");
```

不过，您不能像这样折行：

```
document.write \
("Hello World!");
```

您知道吗？

提示：JavaScript 是脚本语言。浏览器会在读取代码时，逐行地执行脚本代码。而对于传统编程来说，会在执行前对所有代码进行编译。

课外阅读

JavaScript 高级教程： ECMAScript 语法

JavaScript 注释

JavaScript 注释可用于提高代码的可读性。

JavaScript 注释

JavaScript 不会执行注释。

我们可以添加注释来对 JavaScript 进行解释，或者提高代码的可读性。

单行注释以 // 开头。

例子

下面的例子使用单行注释来解释代码：

```
// 输出标题：
document.getElementById("myH1").innerHTML="Welcome to my Homepage";
// 输出段落：
document.getElementById("myP").innerHTML="This is my first paragraph.";
```

JavaScript 多行注释

多行注释以 / 开始，以 / 结尾。

下面的例子使用多行注释来解释代码：

例子

```
/*
下面的这些代码会输出
一个标题和一个段落
并将代表主页的开始
*/
document.getElementById("myH1").innerHTML="Welcome to my Homepage";
document.getElementById("myP").innerHTML="This is my first paragraph.";
```

使用注释来阻止执行

例子 1

在下面的例子中，注释用于阻止其中一条代码行的执行（可用于调试）：

```
//document.getElementById("myH1").innerHTML="Welcome to my Homepage";
document.getElementById("myP").innerHTML="This is my first paragraph.";
```

例子 2

在下面的例子中，注释用于阻止代码块的执行（可用于调试）：

```
/*
document.getElementById("myH1").innerHTML="Welcome to my Homepage";
document.getElementById("myP").innerHTML="This is my first paragraph.";
*/
```

在行末使用注释

在下面的例子中，我们把注释放到代码行的结尾处：

例子

```
var x=5;    // 声明 x 并把 5 赋值给它
var y=x+2;  // 声明 y 并把 x+2 赋值给它
```

课外阅读

JavaScript 高级教程： ECMAScript 语法

JavaScript 变量

变量是存储信息的容器。

实例

```
var x=2;
var y=3;
var z=x+y;
```

就像代数那样

```
x=2
y=3
z=x+y
```

在代数中，我们使用字母（比如 x）来保存值（比如 2）。

通过上面的表达式 z=x+y，我们能够计算出 z 的值为 5。

在 JavaScript 中，这些字母被称为变量。

提示：您可以把变量看做存储数据的容器。

JavaScript 变量

与代数一样，JavaScript 变量可用于存放值（比如 x=2）和表达式（比如 z=x+y）。

变量可以使用短名称（比如 x 和 y），也可以使用描述性更好的名称（比如 age, sum, totalvolume）。

- 变量必须以字母开头
- 变量也能以 $ 和 _ 符号开头（不过我们不推荐这么做）
- 变量名称对大小写敏感（y 和 Y 是不同的变量）

提示：JavaScript 语句和 JavaScript 变量都对大小写敏感。

JavaScript 数据类型

JavaScript 变量还能保存其他数据类型，比如文本值 (name="Bill Gates")。

在 JavaScript 中，类似 "Bill Gates" 这样一条文本被称为字符串。

JavaScript 变量有很多种类型，但是现在，我们只关注数字和字符串。

当您向变量分配文本值时，应该用双引号或单引号包围这个值。

当您向变量赋的值是数值时，不要使用引号。如果您用引号包围数值，该值会被作为文本来处理。

例子

```
var pi=3.14;
var name="Bill Gates";
var answer='Yes I am!';
```

声明（创建） JavaScript 变量

在 JavaScript 中创建变量通常称为“声明”变量。

我们使用 var 关键词来声明变量：

```
var carname;
```

变量声明之后，该变量是空的（它没有值）。

如需向变量赋值，请使用等号：

```
carname="Volvo";
```

不过，您也可以在声明变量时对其赋值：

```
var carname="Volvo";
```

例子

在下面的例子中，我们创建了名为 carname 的变量，并向其赋值 "Volvo"，然后把它放入 id="demo" 的 HTML 段落中：

```
<p id="demo"></p>
var carname="Volvo";
document.getElementById("demo").innerHTML=carname;
```

提示：一个好的编程习惯是，在代码开始处，统一对需要的变量进行声明。

一条语句，多个变量

您可以在一条语句中声明很多变量。该语句以 var 开头，并使用逗号分隔变量即可：

```
var name="Gates", age=56, job="CEO";
```

声明也可横跨多行：

```
var name="Gates",
age=56,
job="CEO";
```

Value = undefined

在计算机程序中，经常会声明无值的变量。未使用值来声明的变量，其值实际上是 undefined。

在执行过以下语句后，变量 carname 的值将是 undefined：

```
var carname;
```

重新声明 JavaScript 变量

如果重新声明 JavaScript 变量，该变量的值不会丢失：

在以下两条语句执行后，变量 carname 的值依然是 "Volvo"：

```
var carname="Volvo";
var carname;
```

JavaScript 算数

您可以通过 JavaScript 变量来做算数，使用的是 = 和 + 这类运算符：

例子

```
y=5;
x=y+2;
```

您将在本教程稍后的章节学到更多有关 JavaScript 运算符的知识。

课外阅读

JavaScript 高级教程： ECMAScript 语法、 ECMAScript 变量

JavaScript 数据类型

字符串、数字、布尔、数组、对象、**Null**、**Undefined**

JavaScript 拥有动态类型

JavaScript 拥有动态类型。这意味着相同的变量可用作不同的类型：

实例

```
var x                // x 为 undefined
var x = 6;           // x 为数字
var x = "Bill";      // x 为字符串
```

JavaScript 字符串

字符串是存储字符（比如 "Bill Gates"）的变量。

字符串可以是引号中的任意文本。您可以使用单引号或双引号：

实例

```
var carname="Bill Gates";
var carname='Bill Gates';
```

您可以在字符串中使用引号，只要不匹配包围字符串的引号即可：

实例

```
var answer="Nice to meet you!";
var answer="He is called 'Bill'";
var answer='He is called "Bill"';
```

您将在本教程的高级部分学到更多关于字符串的知识。

JavaScript 数字

JavaScript 只有一种数字类型。数字可以带小数点，也可以不带：

实例

```
var x1=34.00;      //使用小数点来写
var x2=34;         //不使用小数点来写
```

极大或极小的数字可以通过科学（指数）计数法来书写：

实例

```
var y=123e5;      // 12300000
var z=123e-5;     // 0.00123
```

您将在本教程的高级部分学到更多关于数字的知识。

JavaScript 布尔

布尔（逻辑）只能有两个值：true 或 false。

```
var x=true
var y=false
```

布尔常用在条件测试中。您将在本教程稍后的章节中学到更多关于条件测试的知识。

JavaScript 数组

下面的代码创建名为 cars 的数组：

```
var cars=new Array();
cars[0]="Audi";
cars[1]="BMW";
cars[2]="Volvo";
```

或者 (condensed array):

```
var cars=new Array("Audi","BMW","Volvo");
```

或者 (literal array):

实例

```
var cars=["Audi","BMW","Volvo"];
```

数组下标是基于零的，所以第一个项目是 [0]，第二个是 [1]，以此类推。

您将在本教程稍后的章节中学到更多关于数组的知识。

JavaScript 对象

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔：

```
var person={firstname:"Bill", lastname:"Gates", id:5566};
```

上面例子中的对象 (person) 有三个属性：firstname、lastname 以及 id。

空格和折行无关紧要。声明可横跨多行：

```
var person={
firstname : "Bill",
lastname  : "Gates",
id        :  5566
};
```

对象属性有两种寻址方式：

实例

```
name=person.lastname;
name=person["lastname"];
```

您将在本教程稍后的章节中学到更多关于对象的知识。

Undefined 和 Null

Undefined 这个值表示变量不含有值。

可以通过将变量的值设置为 null 来清空变量。

实例

```
cars=null;
person=null;
```

声明变量类型

当您声明新变量时，可以使用关键词 "new" 来声明其类型：

```
var carname=new String;
var x=      new Number;
var y=      new Boolean;
var cars=   new Array;
var person= new Object;
```

JavaScript 变量均为对象。当您声明一个变量时，就创建了一个新的对象。

课外阅读

JavaScript 高级教程：

JavaScript 对象

JavaScript 中的所有事物都是对象：字符串、数字、数组、日期，等等。

在 **JavaScript** 中，对象是拥有属性和方法的数据。

属性和方法

属性是与对象相关的值。

方法是能够在对象上执行的动作。

举例：汽车就是现实生活中的对象。

汽车的属性：

```
car.name=Fiat

car.model=500

car.weight=850kg

car.color=white
```

汽车的方法：

```
car.start()

car.drive()

car.brake()
```

汽车的属性包括名称、型号、重量、颜色等。

所有汽车都有这些属性，但是每款车的属性都不尽相同。

汽车的方法可以是启动、驾驶、刹车等。

所有汽车都拥有这些方法，但是它们被执行的时间都不尽相同。

JavaScript 中的对象

在 JavaScript 中，对象是数据（变量），拥有属性和方法。

当您像这样声明一个 JavaScript 变量时：

```
var txt = "Hello";
```

您实际上已经创建了一个 JavaScript 字符串对象。字符串对象拥有内建的属性 length。对于上面的字符串来说，length 的值是 5。字符串对象同时拥有若干个内建的方法。

属性：

```
txt.length=5
```

方法：

```
txt.indexOf()

txt.replace()

txt.search()
```

提示：在面向对象的语言中，属性和方法常被称为对象的成员。

在本教程稍后的章节中，您将学到有关字符串对象的更多属性和方法。

创建 JavaScript 对象

JavaScript 中的几乎所有事务都是对象：字符串、数字、数组、日期、函数，等等。

你也可以创建自己的对象。

本例创建名为 "person" 的对象，并为其添加了四个属性：

实例

```
person=new Object();
person.firstname="Bill";
person.lastname="Gates";
person.age=56;
person.eyecolor="blue";
```

创建新 JavaScript 对象有很多不同的方法，并且您还可以向已存在的对象添加属性和方法。

您将在本教程稍后的章节学到更多相关的内容。

访问对象的属性

访问对象属性的语法是：

```
_objectName_._propertyName_
```

本例使用 String 对象的 length 属性来查找字符串的长度：

```
var message="Hello World!";
var x=message.length;
```

在以上代码执行后，x 的值是：

```
12
```

访问对象的方法

您可以通过下面的语法调用方法：

```
_objectName_._methodName_()
```

这个例子使用 String 对象的 toUpperCase() 方法来把文本转换为大写：

```
var message="Hello world!";
var x=message.toUpperCase();
```

在以上代码执行后，x 的值是：

```
HELLO WORLD!
```

您知道吗？

提示：在面向对象的语言中，使用 camel-case 标记法的函数是很常见的。您会经常看到 someMethod() 这样的函数名，而不是 some_method()。

课外书

如需更多有关 *JavaScript* 对象的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 面向对象技术

本节简要介绍了面向对象技术的术语、面向对象语言的要求以及对象的构成。

ECMAScript 对象应用

本节讲解了如何声明和实例化对象，如何引用和废除对象，以及绑定的概念。

ECMAScript 对象类型

本节介绍了 ECMAScript 的三种类型：本地对象、内置对象和宿主对象，并提供了指向相关参考手册的链接。

ECMAScript 对象作用域

本节讲解了 ECMAScript 作用域以及 this 关键字。

ECMAScript 定义类或对象

本节详细讲解了创建 ECMAScript 对象或类的各种方式。

ECMAScript 修改对象

本节讲解了如何通过创建新方法或重定义已有方法来修改对象。

JavaScript 函数

函数是由事件驱动的或者当它被调用时执行的可重复使用的代码块。

实例

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction()
{
alert("Hello World!");
}
</script>
</head>

<body>
<button onclick="myFunction()">点击这里</button>
</body>
</html>
```

JavaScript 函数语法

函数就是包裹在花括号中的代码块，前面使用了关键词 function：

```
function functionname()
{
这里是要执行的代码
}
```

当调用该函数时，会执行函数内的代码。

可以在某事件发生时直接调用函数（比如当用户点击按钮时），并且可由 JavaScript 在任何位置进行调用。

提示：JavaScript 对大小写敏感。关键词 function 必须是小写的，并且必须以与函数名称相同的大小写来调用函数。

调用带参数的函数

在调用函数时，您可以向其传递值，这些值被称为参数。

这些参数可以在函数中使用。

您可以发送任意多的参数，由逗号 (,) 分隔：

```
myFunction(_argument1_,_argument2_)
```

当您声明函数时，请把参数作为变量来声明：

```
function myFunction(`var1`,`var2`)
{
这里是要执行的代码
}
```

变量和参数必须以一致的顺序出现。第一个变量就是第一个被传递的参数的给定的值，以此类推。

实例

```
<button onclick="myFunction('Bill Gates','CEO')">点击这里</button>

<script>
function myFunction(name,job)
{
alert("Welcome " + `name` + ", the " + `job`);
}
</script>
```

上面的函数会当按钮被点击时提示 "Welcome Bill Gates, the CEO"。

函数很灵活，您可以使用不同的参数来调用该函数，这样就会给出不同的消息：

实例

```
<button onclick="myFunction('Harry Potter','Wizard')">点击这里</button>
<button onclick="myFunction('Bob','Builder')">点击这里</button>
```

根据您点击的不同的按钮，上面的例子会提示 "Welcome Harry Potter, the Wizard" 或 "Welcome Bob, the Builder"。

带有返回值的函数

有时，我们会希望函数将值返回调用它的地方。

通过使用 return 语句就可以实现。

在使用 return 语句时，函数会停止执行，并返回指定的值。

语法

```
function myFunction()
{
var x=5;
`return x;`
}
```

上面的函数会返回值 5。

注释：整个 JavaScript 并不会停止执行，仅仅是函数。JavaScript 将继续执行代码，从调用函数的地方。

函数调用将被返回值取代：

```
var myVar=myFunction();
```

myVar 变量的值是 5，也就是函数 "myFunction()" 所返回的值。

即使不把它保存为变量，您也可以使用返回值：

```
document.getElementById("demo").innerHTML=myFunction();
```

"demo" 元素的 innerHTML 将成为 5，也就是函数 "myFunction()" 所返回的值。

您可以使返回值基于传递到函数中的参数：

实例

计算两个数字的乘积，并返回结果：

```
function myFunction(a,b)
{
return a*b;
}

document.getElementById("demo").innerHTML=myFunction(4,3);
```

"demo" 元素的 innerHTML 将是：

```
12
```

在您仅仅希望退出函数时，也可使用 return 语句。返回值是可选的：

```
function myFunction(a,b)
{
if (a>b)
  {
  return;
  }
x=a+b
}
```

如果 a 大于 b，则上面的代码将退出函数，并不会计算 a 和 b 的总和。

局部 JavaScript 变量

在 JavaScript 函数内部声明的变量（使用 var）是局部变量，所以只能在函数内部访问它。（该变量的作用域是局部的）。

您可以在不同的函数中使用名称相同的局部变量，因为只有声明过该变量的函数才能识别出该变量。

只要函数运行完毕，本地变量就会被删除。

全局 JavaScript 变量

在函数外声明的变量是全局变量，网页上的所有脚本和函数都能访问它。

JavaScript 变量的生存期

JavaScript 变量的生命期从它们被声明的时间开始。

局部变量会在函数运行以后被删除。

全局变量会在页面关闭后被删除。

向未声明的 JavaScript 变量来分配值

如果您把值赋给尚未声明的变量，该变量将被自动作为全局变量声明。

这条语句：

```
carname="Volvo";
```

将声明一个全局变量 carname，即使它在函数内执行。

课外书

如需更多有关 *JavaScript* 函数的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 函数概述

本节讲解函数的概念，ECMAScript 如何声明并调用函数，以及函数如何返回值。

ECMAScript arguments 对象

本节介绍了此对象的基本用法，然后讲解了如何使用 length 属性来测定函数的参数数目，以及模拟函数重载。

ECMAScript Function 对象（类）

本节讲解了如何使用 Function 类创建函数，然后介绍了 Function 对象的属性和方法。

ECMAScript 闭包（closure）

本节讲解了闭包（closure）的概念，并分别为您展示了简单和稍复杂的两个闭包实例。

JavaScript 运算符

运算符 **=** 用于赋值。

运算符 **+** 用于加值。

运算符 = 用于给 JavaScript 变量赋值。

算术运算符 + 用于把值加起来。

```
y=5;
z=2;
x=y+z;
```

在以上语句执行后，x 的值是 7。

JavaScript 算术运算符

算术运算符用于执行变量与/或值之间的算术运算。

给定 *y=5*，下面的表格解释了这些算术运算符：

运算符	描述	例子	结果
+	加	x=y+2	x=7
-	减	x=y-2	x=3
*	乘	x=y*2	x=10
/	除	x=y/2	x=2.5
%	求余数 (保留整数)	x=y%2	x=1
++	累加	x=++y	x=6
--	递减	x=--y	x=4

JavaScript 赋值运算符

赋值运算符用于给 JavaScript 变量赋值。

给定 *x=10* 和 *y=5*，下面的表格解释了赋值运算符：

运算符	例子	等价于	结果
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

用于字符串的 + 运算符

- 运算符用于把文本值或字符串变量加起来（连接起来）。

如需把两个或多个字符串变量连接起来，请使用 + 运算符。

```
txt1="What a very";
txt2="nice day";
txt3=txt1+txt2;
```

在以上语句执行后，变量 txt3 包含的值是 "What a verynice day"。

要想在两个字符串之间增加空格，需要把空格插入一个字符串之中：

```
txt1="What a very ";
txt2="nice day";
txt3=txt1+txt2;
```

或者把空格插入表达式中：

```
txt1="What a very";
txt2="nice day";
txt3=txt1+" "+txt2;
```

在以上语句执行后，变量 txt3 包含的值是：

"What a very nice day"

对字符串和数字进行加法运算

请看这些例子：

```
x=5+5;
document.write(x);

x="5"+"5";
document.write(x);

x=5+"5";
document.write(x);

x="5"+5;
document.write(x);
```

TIY

规则是：

如果把数字与字符串相加，结果将成为字符串。

课外书

如需更多有关 *JavaScript* 运算符的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 一元运算符

一元运算符只有一个参数，即要操作的对象或值。本节讲解 ECMAScript 中最简单的运算符 - 一元运算符。

ECMAScript 位运算符

位运算符是在数字底层进行操作的。本节深入讲解了有关整数的知识，并介绍了 ECMAScript 的各种位运算符。

ECMAScript Boolean 运算符

Boolean 运算符非常重要。本节深入讲解三种 Boolean 运算符：NOT、AND 和 OR。

ECMAScript 乘性运算符

本节讲解 ECMAScript 的乘性运算符：乘法、除法、取模运算符，以及它们的特殊行为。

ECMAScript 加性运算符

本节讲解 ECMAScript 的加性运算符：加法、减法运算符，以及它们的特殊行为。

ECMAScript 关系运算符

关系运算符执行的是比较运算。本节讲解关系运算符的常规比较方式，以及如何比较字符串与数字。

ECMAScript 等性运算符

等性运算符用于判断变量是否相等。ECMAScript 提供两套等性运算符：等号和非等号，以及全等号和非全等号。

ECMAScript 条件运算符

本节讲解 ECMAScript 中的条件运算符。

ECMAScript 赋值运算符

本节讲解 ECMAScript 中的赋值运算符。

ECMAScript 逗号运算符符

本节讲解 ECMAScript 中的逗号运算符。

JavaScript 比较和逻辑运算符

比较和逻辑运算符用于测试 **true** 或 **false**。

比较运算符

比较运算符在逻辑语句中使用，以测定变量或值是否相等。

给定 x=5，下面的表格解释了比较运算符：

运算符	描述	例子
==	等于	x==8 为 false
===	全等（值和类型）	x===5 为 true；x==="5" 为 false
!=	不等于	x!=8 为 true
>	大于	x>8 为 false
<	小于	x<8 为 true
>=	大于或等于	x>=8 为 false
<=	小于或等于	x<=8 为 true

如何使用

可以在条件语句中使用比较运算符对值进行比较，然后根据结果来采取行动：

```
if (age<18) document.write("Too young");
```

您将在本教程的下一节中学习更多有关条件语句的知识。

逻辑运算符

逻辑运算符用于测定变量或值之间的逻辑。

给定 x=6 以及 y=3，下表解释了逻辑运算符：

运算符	描述	例子
&&	and	(x < 10 && y > 1) 为 true
&124;&124;	or	(x==5 &124;&124; y==5) 为 false
!	not	!(x==y) 为 true

条件运算符

JavaScript 还包含了基于某些条件对变量进行赋值的条件运算符。

语法

```
variablename=(condition)?value1:value2
```

例子

```
greeting=(visitor=="PRES")?"Dear President ":"Dear ";
```

如果变量 visitor 中的值是 "PRES"，则向变量 greeting 赋值 "Dear President "，否则赋值 "Dear"。

JavaScript If...Else 语句

条件语句用于基于不同的条件来执行不同的动作。

条件语句

通常在写代码时，您总是需要为不同的决定来执行不同的动作。您可以在代码中使用条件语句来完成该任务。

在 JavaScript 中，我们可使用以下条件语句：

- *if* 语句 - 只有当指定条件为 true 时，使用该语句来执行代码
- *if...else* 语句 - 当条件为 true 时执行代码，当条件为 false 时执行其他代码
- *if...else if....else* 语句 - 使用该语句来选择多个代码块之一来执行
- *switch* 语句 - 使用该语句来选择多个代码块之一来执行

If 语句

只有当指定条件为 true 时，该语句才会执行代码。

语法

```
if (条件)
  {
  只有当条件为 true 时执行的代码
  }
```

注意：请使用小写的 if。使用大写字母（IF）会生成 JavaScript 错误！

实例

当时间小于 20:00 时，生成一个“Good day”问候：

```
if (time<20)
  {
  x="Good day";
  }
```

x 的结果是：

```
 <script>var d=new Date();
var time=d.getHours();
if (time<20)
  {
  document.write("Good day");
  }</script> Good day
```

请注意，在这个语法中，没有 ..else..。您已经告诉浏览器只有在指定条件为 true 时才执行代码。

If...else 语句

请使用 if....else 语句在条件为 true 时执行代码，在条件为 false 时执行其他代码。

语法

```
if (条件)
  {
  当条件为 true 时执行的代码
  }
else
  {
  当条件不为 true 时执行的代码
  }
```

实例

当时间小于 20:00 时，将得到问候 "Good day"，否则将得到问候 "Good evening"。

```
if (time<20)
  {
  x="Good day";
  }
else
  {
  x="Good evening";
  }
```

x 的结果是：

```
 <script>var d=new Date();
var time=d.getHours();
if (time<20)
  {
  document.write("Good day");
  }
else
  {
  document.write("Good evening");
  }</script> Good day
```

If...else if...else 语句

使用 if....else if...else 语句来选择多个代码块之一来执行。

语法

```
if (条件 1)
  {
  当条件 1 为 true 时执行的代码
  }
else if (条件 2)
  {
  当条件 2 为 true 时执行的代码
  }
else
  {
  当条件 1 和 条件 2 都不为 true 时执行的代码
  }
```

实例

如果时间小于 10:00，则将发送问候 "Good morning"，否则如果时间小于 20:00，则发送问候 "Good day"，否则发送问候 "Good evening"：

```
if (time<10)
  {
  x="Good morning";
  }
else if (time<20)
  {
  x="Good day";
  }
else
  {
  x="Good evening";
  }
```

x 的结果是：

```
 <script>var d=new Date();
var time=d.getHours();
if (time<10)
  {
  document.write("Good morning");
  }
else if (time<20)
  {
  document.write("Good day");
  }
else
  {
  document.write("Good evening");
  }</script> Good day
```

更多实例

随机的链接

本例将输出 W3School 或微软公司的链接。通过使用随机数，每个链接被输出的机会为 50%。

课外书

如需更多有关 *JavaScript if* 语句的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript if 语句

if 语句是 ECMAScript 中最常用的语句之一。本节为您详细讲解了如何使用 if 语句。

JavaScript Switch 语句

switch 语句用于基于不同的条件来执行不同的动作。

JavaScript Switch 语句

请使用 switch 语句来选择要执行的多个代码块之一。

语法

```
switch(n)
{
case 1:
  执行代码块 1
  break;
case 2:
  执行代码块 2
  break;
default:
  n 与 case 1 和 case 2 不同时执行的代码
}
```

工作原理：首先设置表达式 n（通常是一个变量）。随后表达式的值会与结构中的每个 case 的值做比较。如果存在匹配，则与该 case 关联的代码块会被执行。请使用 *break* 来阻止代码自动地向下一个 case 运行。

实例

显示今日的周名称。请注意 Sunday=0, Monday=1, Tuesday=2, 等等：

```
var day=new Date().getDay();
switch (day)
{
case 0:
  x="Today it's Sunday";
  break;
case 1:
  x="Today it's Monday";
  break;
case 2:
  x="Today it's Tuesday";
  break;
case 3:
  x="Today it's Wednesday";
  break;
case 4:
  x="Today it's Thursday";
  break;
case 5:
  x="Today it's Friday";
  break;
case 6:
  x="Today it's Saturday";
  break;
}
```

x 的结果：

```
 <script>var d=new Date().getDay();
switch (d)
{
case 0:
  x="Today it's Sunday";
  break;
case 1:
  x="Today it's Monday";
  break;
case 2:
  x="Today it's Tuesday";
  break;
case 3:
  x="Today it's Wednesday";
  break;
case 4:
  x="Today it's Thursday";
  break;
case 5:
  x="Today it's Friday";
  break;
case 6:
  x="Today it's Saturday";
  break;
}
document.write(x);</script> Today it's Tuesday
```

default 关键词

请使用 default 关键词来规定匹配不存在时做的事情：

实例

如果今天不是周六或周日，则会输出默认的消息：

```
var day=new Date().getDay();
switch (day)
{
case 6:
  x="Today it's Saturday";
  break;
case 0:
  x="Today it's Sunday";
  break;
`default:
  x="Looking forward to the Weekend";`
}
```

x 的结果：

```
 <script>var d=new Date().getDay();
switch (d)
{
case 6:
  x="Today it's Saturday";
  break;
case 0:
  x="Today it's Sunday";
  break;
default:
  x="Looking forward to the Weekend";
}
document.write(x);</script> Looking forward to the Weekend
```

课外书

如需更多有关 *JavaScript Switch* 语句的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript switch 语句

switch 语句是 if 语句的兄弟语句。本节介绍了 switch 语句的用法，以及与 Java 中的 switch 语句的不同。

JavaScript For 循环

循环可以将代码块执行指定的次数。

JavaScript 循环

如果您希望一遍又一遍地运行相同的代码，并且每次的值都不同，那么使用循环是很方便的。

我们可以这样输出数组的值：

```
document.write(cars[0] + "<br>");
document.write(cars[1] + "<br>");
document.write(cars[2] + "<br>");
document.write(cars[3] + "<br>");
document.write(cars[4] + "<br>");
document.write(cars[5] + "<br>");
```

不过通常我们这样写：

```
for (var i=0;i<cars.length;i++)
{
document.write(cars[i] + "<br>");
}
```

不同类型的循环

JavaScript 支持不同类型的循环：

- *for* - 循环代码块一定的次数
- *for/in* - 循环遍历对象的属性
- *while* - 当指定的条件为 true 时循环指定的代码块
- *do/while* - 同样当指定的条件为 true 时循环指定的代码块

For 循环

for 循环是您在希望创建循环时常会用到的工具。

下面是 for 循环的语法：

```
for (语句 1; 语句 2; 语句 3)
  {
  被执行的代码块
  }
```

语句 *1* 在循环（代码块）开始前执行

语句 *2* 定义运行循环（代码块）的条件

语句 *3* 在循环（代码块）已被执行之后执行

实例

```
for (var i=0; i<5; i++)
  {
  x=x + "The number is " + i + "<br>";
  }
```

从上面的例子中，您可以看到：

Statement 1 在循环开始之前设置变量 (var i=0)。

Statement 2 定义循环运行的条件（i 必须小于 5）。

Statement 3 在每次代码块已被执行后增加一个值 (i++)。

语句 1

通常我们会使用语句 1 初始化循环中所用的变量 (var i=0)。

语句 1 是可选的，也就是说不使用语句 1 也可以。

您可以在语句 1 中初始化任意（或者多个）值：

实例:

```
for (`var i=0,len=cars.length;` i<len; i++)
{
document.write(cars[i] + "<br>");
}
```

同时您还可以省略语句 1（比如在循环开始前已经设置了值时）：

实例:

```
var i=2,len=cars.length;
for (; i<len; i++)
{
document.write(cars[i] + "<br>");
}
```

语句 2

通常语句 2 用于评估初始变量的条件。

语句 2 同样是可选的。

如果语句 2 返回 true，则循环再次开始，如果返回 false，则循环将结束。

提示：如果您省略了语句 2，那么必须在循环内提供 *break*。否则循环就无法停下来。这样有可能令浏览器崩溃。请在本教程稍后的章节阅读有关 break 的内容。

语句 3

通常语句 3 会增加初始变量的值。

语句 3 也是可选的。

语句 3 有多种用法。增量可以是负数 (i--)，或者更大 (i=i+15)。

语句 3 也可以省略（比如当循环内部有相应的代码时）：

实例:

```
var i=0,len=cars.length;
for (; i<len; )
{
document.write(cars[i] + "<br>");
i++;
}
```

For/In 循环

JavaScript for/in 语句循环遍历对象的属性：

实例

```
var person={fname:"John",lname:"Doe",age:25};

for (x `in` person)
  {
  txt=txt + person[x];
  }
```

您将在有关 JavaScript 对象的章节学到更多有关 for / in 循环的知识。

While 循环

我们将在下一章为您讲解 while 循环和 do/while 循环。

课外书

如需更多有关 *JavaScript for* 语句的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 迭代语句

迭代语句又叫循环语句。本节为您介绍 ECMAScript 提供的四种迭代语句。

JavaScript While 循环

只要指定条件为 **true**，循环就可以一直执行代码。

while 循环

While 循环会在指定条件为真时循环执行代码块。

语法

```
while (条件)
  {
  需要执行的代码
  }
```

实例

本例中的循环将继续运行，只要变量 i 小于 5：

```
while (i<5)
  {
  x=x + "The number is " + i + "<br>";
  i++;
  }
```

提示：如果您忘记增加条件中所用变量的值，该循环永远不会结束。该可能导致浏览器崩溃。

do/while 循环

do/while 循环是 while 循环的变体。该循环会执行一次代码块，在检查条件是否为真之前，然后如果条件为真的话，就会重复这个循环。

语法

```
do
  {
  需要执行的代码
  }
while (条件);
```

实例

下面的例子使用 do/while 循环。该循环至少会执行一次，即使条件是 false，隐藏代码块会在条件被测试前执行：

```
do
  {
  x=x + "The number is " + i + "<br>";
  i++;
  }
while (i<5);
```

别忘记增加条件中所用变量的值，否则循环永远不会结束！

比较 for 和 while

如果您已经阅读了前面那一章关于 for 循环的内容，您会发现 while 循环与 for 循环很像。

for 语句实例

本例中的循环使用 for 循环来显示 cars 数组中的所有值：

```
cars=["BMW","Volvo","Saab","Ford"];
var i=0;
for (;cars[i];)
{
document.write(cars[i] + "<br>");
i++;
}
```

while 语句实例

本例中的循环使用使用 while 循环来显示 cars 数组中的所有值：

```
cars=["BMW","Volvo","Saab","Ford"];
var i=0;
while (cars[i])
{
document.write(cars[i] + "<br>");
i++;
}
```

课外书

如需更多有关 *JavaScript while* 语句的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 迭代语句

迭代语句又叫循环语句。本节为您介绍 ECMAScript 提供的四种迭代语句。

JavaScript Break 和 Continue 语句

break 语句用于跳出循环。

continue 用于跳过循环中的一个迭代。

Break 语句

我们已经在本教程稍早的章节中见到过 break 语句。它用于跳出 switch() 语句。

break 语句可用于跳出循环。

break 语句跳出循环后，会继续执行该循环之后的代码（如果有的话）：

实例

```
for (i=0;i<10;i++)
  {
  if (i==3)
    {
    `break;`
    }
  x=x + "The number is " + i + "<br>";
  }
```

由于这个 if 语句只有一行代码，所以可以省略花括号：

```
for (i=0;i<10;i++)
  {
  `if (i==3) break;`
  x=x + "The number is " + i + "<br>";
  }
```

Continue 语句

continue 语句中断循环中的迭代，如果出现了指定的条件，然后继续循环中的下一个迭代。

该例子跳过了值 3：

实例

```
for (i=0;i<=10;i++)
 {
 if (i==3) continue;
  x=x + "The number is " + i + "<br>";
  }
```

JavaScript 标签

正如您在 switch 语句那一章中看到的，可以对 JavaScript 语句进行标记。

如需标记 JavaScript 语句，请在语句之前加上冒号：

```
label:
语句
```

break 和 continue 语句仅仅是能够跳出代码块的语句。

语法

```
break labelname;

continue labelname;
```

continue 语句（带有或不带标签引用）只能用在循环中。

break 语句（不带标签引用），只能用在循环或 switch 中。

通过标签引用，break 语句可用于跳出任何 JavaScript 代码块：

实例

```
cars=["BMW","Volvo","Saab","Ford"];
list:
{
document.write(cars[0] + "<br>");
document.write(cars[1] + "<br>");
document.write(cars[2] + "<br>");
break list;
document.write(cars[3] + "<br>");
document.write(cars[4] + "<br>");
document.write(cars[5] + "<br>");
}
```

课外书

如需更多有关 *JavaScript Break* 和 *Continue* 语句的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript break 和 continue 语句

本节讲解了break 语句和 continue 语句的不同之处，以及如何与有标签语句一起使用。

JavaScript 错误 - Throw、Try 和 Catch

try 语句测试代码块的错误。

catch 语句处理错误。

throw 语句创建自定义错误。

错误一定会发生

当 JavaScript 引擎执行 JavaScript 代码时，会发生各种错误：

可能是语法错误，通常是程序员造成的编码错误或错别字。

可能是拼写错误或语言中缺少的功能（可能由于浏览器差异）。

可能是由于来自服务器或用户的错误输出而导致的错误。

当然，也可能是由于许多其他不可预知的因素。

JavaScript 抛出错误

当错误发生时，当事情出问题时，JavaScript 引擎通常会停止，并生成一个错误消息。

描述这种情况的技术术语是：JavaScript 将抛出一个错误。

JavaScript 测试和捕捉

try 语句允许我们定义在执行时进行错误测试的代码块。

catch 语句允许我们定义当 try 代码块发生错误时，所执行的代码块。

JavaScript 语句 *try* 和 *catch* 是成对出现的。

语法

```
try
  {
  //在这里运行代码
  }
catch(err)
  {
  //在这里处理错误
  }
```

实例

在下面的例子中，我们故意在 try 块的代码中写了一个错字。

catch 块会捕捉到 try 块中的错误，并执行代码来处理它。

```
<!DOCTYPE html>
<html>
<head>
<script>
var txt="";
function message()
{
try
  {
  adddlert("Welcome guest!");
  }
catch(err)
  {
  txt="There was an error on this page.\n\n";
  txt+="Error description: " + err.message + "\n\n";
  txt+="Click OK to continue.\n\n";
  alert(txt);
  }
}
</script>
</head>

<body>
<input type="button" value="View message" onclick="message()">
</body>

</html>
```

Throw 语句

throw 语句允许我们创建自定义错误。

正确的技术术语是：创建或抛出异常（exception）。

如果把 throw 与 try 和 catch 一起使用，那么您能够控制程序流，并生成自定义的错误消息。

语法

```
throw _exception_
```

异常可以是 JavaScript 字符串、数字、逻辑值或对象。

实例

本例检测输入变量的值。如果值是错误的，会抛出一个异常（错误）。catch 会捕捉到这个错误，并显示一段自定义的错误消息：

```
<script>
function myFunction()
{
try
  {
  var x=document.getElementById("demo").value;
  if(x=="")     `throw "empty"`;
  if(isNaN(x)) `throw "not a number"`;
  if(x>10)      `throw "too high"`;
  if(x<5)       `throw "too low"`;
  }
catch(err)
  {
  var y=document.getElementById("mess");
  y.innerHTML="Error: " + err + ".";
  }
}
</script>

<h1>My First JavaScript</h1>
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="mess"></p>
```

请注意，如果 getElementById 函数出错，上面的例子也会抛出一个错误。

课外阅读

JavaScript 高级教程： JavaScript 历史 、 JavaScript 实现

JavaScript 表单验证

JavaScript 可用来在数据被送往服务器前对 **HTML** 表单中的这些输入数据进行验证。

JavaScript 表单验证

JavaScript 可用来在数据被送往服务器前对 HTML 表单中的这些输入数据进行验证。

被 JavaScript 验证的这些典型的表单数据有：

- 用户是否已填写表单中的必填项目？
- 用户输入的邮件地址是否合法？
- 用户是否已输入合法的日期？
- 用户是否在数据域 (numeric field) 中输入了文本？

必填（或必选）项目

下面的函数用来检查用户是否已填写表单中的必填（或必选）项目。假如必填或必选项为空，那么警告框会弹出，并且函数的返回值为 false，否则函数的返回值则为 true（意味着数据没有问题）：

```
function validate_required(field,alerttxt)
{
with (field)
{
if (value==null||value=="")
  {alert(alerttxt);return false}
else {return true}
}
}
```

下面是连同 HTML 表单的代码：

```
<html>
<head>
<script type="text/javascript">
 `function validate_required(field,alerttxt)
{
with (field)
  {
  if (value==null||value=="")
    {alert(alerttxt);return false}
  else {return true}
  }
}

function validate_form(thisform)
{
with (thisform)
  {
  if (validate_required(email,"Email must be filled out!")==false)
    {email.focus();return false}
  }
}`
</script>
</head>

<body>
<form action="submitpage.htm" onsubmit="return validate_form(this)" method="post">
Email: <input type="text" name="email" size="30">
<input type="submit" value="Submit">
</form>
</body>

</html>
```

E-mail 验证

下面的函数检查输入的数据是否符合电子邮件地址的基本语法。

意思就是说，输入的数据必须包含 @ 符号和点号(.)。同时，@ 不可以是邮件地址的首字符，并且 @ 之后需有至少一个点号：

```
function validate_email(field,alerttxt)
{
with (field)
{
apos=value.indexOf("@")
dotpos=value.lastIndexOf(".")
if (apos<1||dotpos-apos<2)
  {alert(alerttxt);return false}
else {return true}
}
}
```

下面是连同 HTML 表单的完整代码：

```
<html>
<head>
<script type="text/javascript">
`function validate_email(field,alerttxt)
{
with (field)
{
apos=value.indexOf("@")
dotpos=value.lastIndexOf(".")
if (apos<1||dotpos-apos<2)
  {alert(alerttxt);return false}
else {return true}
}
}

function validate_form(thisform)
{
with (thisform)
{
if (validate_email(email,"Not a valid e-mail address!")==false)
  {email.focus();return false}
}
}`
</script>
</head>

<body>
<form action="submitpage.htm"onsubmit="return validate_form(this);" method="post">
Email: <input type="text" name="email" size="30">
<input type="submit" value="Submit">
</form>
</body>

</html>
```

JavaScript 保留关键字

在 JavaScript 中，一些标识符是保留关键字，不能用作变量名或函数名。

JavaScript 标准

所有的现代浏览器完全支持 ECMAScript 3（ES3，JavaScript 的第三版，从 1999 年开始）。

ECMAScript 4（ES4）未通过。

ECMAScript 5（ES5，2009 年发布），是 JavaScript 最新的官方版本。

随着时间的推移，我们开始看到，所有的现代浏览器已经完全支持 ES5。

JavaScript 保留关键字

Javascript 的保留关键字不可以用作变量、标签或者函数名。有些保留关键字是作为 Javascript 以后扩展使用。

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

- 标记的关键字是 ECMAScript5 中新添加的。

JavaScript 对象、属性和方法

您也应该避免使用 JavaScript 内置的对象、属性和方法的名称作为 Javascript 的变量或函数名：

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

Java 保留关键字

JavaScript 经常与 Java 一起使用。您应该避免使用一些 Java 对象和属性作为 JavaScript 标识符：

getClass	java	JavaArray
javaClass	JavaObject	JavaPackage

Windows 保留关键字

JavaScript 可以在 HTML 外部使用。它可在许多其他应用程序中作为编程语言使用。

在 HTML 中，您必须（为了可移植性，您也应该这么做）避免使用 HTML 和 Windows 对象和属性的名称作为 Javascript 的变量及函数名：

alert	all	anchor	anchors	
assign	blur	button	checkbox	c
clearTimeout	clientInformation	close	closed	c
constructor	crypto	decodeURI	decodeURIComponent	c
document	element	elements	embed	e
encodeURI	encodeURIComponent	escape	event	f
focus	form	forms	frame	i
innerWidth	layer	layers	link	l
mimeTypes	navigate	navigator	frames	f
hidden	history	image	images	c
open	opener	option	outerHeight	c
packages	pageXOffset	pageYOffset	parent	p
parseInt	password	pkcs11	plugin	p
propertyIsEnum	radio	reset	screenX	s
scroll	secure	select	self	s
setTimeout	status	submit	taint	t
textarea	top	unescape	untaint	v

HTML 事件句柄

除此之外，您还应该避免使用 HTML 事件句柄的名称作为 Javascript 的变量及函数名。

实例：

onblur	onclick	onerror	onfocus
onkeydown	onkeypress	onkeyup	onmouseover
onload	onmouseup	onmousedown	onsubmit

非标准 JavaScript

除了保留关键字，在 JavaScript 实现中也有一些非标准的关键字。

一个实例是 **const** 关键字，用于定义变量。 一些 JavaScript 引擎把 const 当作 var 的同义词。另一些引擎则把 const 当作只读变量的定义。

Const 是 JavaScript 的扩展。JavaScript 引擎支持它用在 Firefox 和 Chrome 中。但是它并不是 JavaScript 标准 ES3 或 ES5 的组成部分。建议：不要使用它。

JavaScript JSON

JSON 是用于存储和传输数据的格式。

JSON 通常用于服务端向网页传递数据 。

什么是 JSON?

- JSON 英文全称 **J**ava**S**cript **O**bject **N**otation
- JSON 是一种轻量级的数据交换格式。
- JSON是独立的语言 *
- JSON 易于理解。

>  JSON 使用 JavaScript 语法，但是 JSON 格式仅仅是一个文本。
> 文本可以被任何编程语言读取及作为数据格式传递。

JSON 实例

以下 JSON 语法定义了 employees 对象: 3 条员工记录（对象）的数组:

JSON Example

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

JSON 格式化后为 JavaScript 对象

JSON 格式在语法上与创建 JavaScript 对象代码是相同的。

由于它们很相似，所以 JavaScript 程序可以很容易的将 JSON 数据转换为 JavaScript 对象。

JSON 语法规则

- 数据为 键/值 对。
- 数据由逗号分隔。

- 大括号保存对象
- 方括号保存数组

JSON 数据 - 一个名称对应一个值

JSON 数据格式为 键/值 对，就像 JavaScript 对象属性。

键/值对包括字段名称（在双引号中），后面一个冒号，然后是值：

```
"firstName":"John"
```

JSON 对象

JSON 对象保存在大括号内。

就像在 JavaScript 中, 对象可以保存多个 键/值 对：

```
{"firstName":"John", "lastName":"Doe"}
```

JSON 数组

JSON 数组保存在中括号内。

就像在 JavaScript 中, 数组可以包含对象：

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

在以上实例中，对象 "employees" 是一个数组。包含了三个对象。

每个为个对象为员工的记录（姓和名）。

JSON 字符串转换为 JavaScript 对象

通常我们从服务器中读取 JSON 数据，并在网页中显示数据。

简单起见，我们网页中直接设置 JSON 字符串 (你还可以阅读我们的 JSON 教程):

首先，创建 JavaScript 字符串，字符传为 JSON 格式的数据：

```
var text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

然后，使用 JavaScript 内置函数 JSON.parse() 将字符串转换为 JavaScript 对象:

```
var obj = JSON.parse(text);
```

最后，在你的页面中使用新的 JavaScript 对象：

实例

```
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```

更多 JSON 信息，你可以阅读我们的 JSON 教程。

javascript:void(0) 含义

我们经常会使用到 javascript:void(0) 这样的代码，那么在 JavaScript 中 javascript:void(0) 代表的是什么意思呢？

javascript:void(0) 中最关键的是 void 关键字， void 是 JavaScript 中非常重要的关键字，该操作符指定要计算一个表达式但是不返回值。

语法格式如下：

```
<head>
<script type="text/javascript">
<!--
void func()
javascript:void func()

或者

void(func())
javascript:void(func())
//-->
</script>
</head>
```

下面的代码创建了一个超级链接，当用户点击以后不会发生任何事。

实例

```
<a href="javascript:void(0)">单击此处什么也不会发生</a>
```

当用户链接时，void(0) 计算为 0，但 Javascript 上没有任何效果。

以下实例中，在用户点击链接后显示警告信息：

实例

```
<head>
<script type="text/javascript">
<!--
//-->
</script>
</head>
<body>
<a href="javascript:void(alert('Warning!!!'))">点我!</a>
</body>
```

以下实例中参数 a 将返回 undefined :

实例

```
<head>
<script type="text/javascript">
<!--
function getValue(){
  var a,b,c;
  a = void ( b = 5, c = 7 );
  document.write('a = ' + a + ' b = ' + b +' c = ' + c );
}
//-->
</script>
</head>
```

href="#"与href="javascript:void(0)"的区别

包含了一个位置信息，默认的锚是**#top** 也就是网页的上端。

而javascript:void(0), 仅仅表示一个死链接。

在页面很长的时候会使用 **#** 来定位页面的具体位置，格式为：**# + id**。

如果你要定义一个死链接请使用 javascript:void(0) 。

实例

```
<a href="javascript:void(0);">点我没有反应的!</a>
<a href="#pos">点我定位到指定位置!</a>
<br><br><br> <p id="pos">尾部定位点</p>
```

JavaScript 高级

JavaScript 对象

JavaScript 中的所有事物都是对象：字符串、数值、数组、函数...

此外，**JavaScript** 允许自定义对象。

JavaScript 对象

JavaScript 提供多个内建对象，比如 String、Date、Array 等等。

对象只是带有属性和方法的特殊数据类型。

访问对象的属性

属性是与对象相关的值。

访问对象属性的语法是：

```
_objectName_._propertyName_
```

这个例子使用了 String 对象的 length 属性来获得字符串的长度：

```
var message="Hello World!";
var x=message.`length`;
```

在以上代码执行后，x 的值将是：

```
12
```

访问对象的方法

方法是能够在对象上执行的动作。

您可以通过以下语法来调用方法：

```
_objectName_._methodName_()
```

这个例子使用了 String 对象的 toUpperCase() 方法来将文本转换为大写：

```
var message="Hello world!";
var x=message.`toUpperCase()`;
```

在以上代码执行后，x 的值将是：

```
HELLO WORLD!
```

创建 JavaScript 对象

通过 JavaScript，您能够定义并创建自己的对象。

创建新对象有两种不同的方法：

1. 定义并创建对象的实例
2. 使用函数来定义对象，然后创建新的对象实例

创建直接的实例

这个例子创建了对象的一个新实例，并向其添加了四个属性：

实例

```
person=new Object();
person.firstname="Bill";
person.lastname="Gates";
person.age=56;
person.eyecolor="blue";
```

替代语法（使用对象 literals）：

实例

```
person={firstname:"John",lastname:"Doe",age:50,eyecolor:"blue"};
```

使用对象构造器

本例使用函数来构造对象：

实例

```
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname;
this.age=age;
this.eyecolor=eyecolor;
}
```

创建 JavaScript 对象实例

一旦您有了对象构造器，就可以创建新的对象实例，就像这样：

```
var myFather=new person("Bill","Gates",56,"blue");
var myMother=new person("Steve","Jobs",48,"green");
```

把属性添加到 JavaScript 对象

您可以通过为对象赋值，向已有对象添加新属性：

假设 personObj 已存在 - 您可以为其添加这些新属性：firstname、lastname、age 以及 eyecolor：

```
person.firstname="Bill";
person.lastname="Gates";
person.age=56;
person.eyecolor="blue";

x=person.firstname;
```

在以上代码执行后，x 的值将是：

```
Bill
```

把方法添加到 JavaScript 对象

方法只不过是附加在对象上的函数。

在构造器函数内部定义对象的方法：

```
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname;
this.age=age;
this.eyecolor=eyecolor;

this.changeName=changeName;
function changeName(name)
{
this.lastname=name;
}
}
```

changeName() 函数 name 的值赋给 person 的 lastname 属性。

现在您可以试一下：

```
myMother.changeName("Ballmer");
```

JavaScript 类

JavaScript 是面向对象的语言，但 JavaScript 不使用类。

在 JavaScript 中，不会创建类，也不会通过类来创建对象（就像在其他面向对象的语言中那样）。

JavaScript 基于 prototype，而不是基于类的。

JavaScript for...in 循环

JavaScript for...in 语句循环遍历对象的属性。

语法

```
for (对象中的变量)
  {
  要执行的代码
  }
```

注释：for...in 循环中的代码块将针对每个属性执行一次。

实例

循环遍历对象的属性：

```
var person={fname:"Bill",lname:"Gates",age:56};

for (x in person)
  {
  txt=txt + person[x];
  }
```

课外书

如需更多有关 *JavaScript* 对象的知识，请阅读 JavaScript 高级教程中的相关内容：

ECMAScript 面向对象技术

本节简要介绍了面向对象技术的术语、面向对象语言的要求以及对象的构成。

ECMAScript 对象应用

本节讲解了如何声明和实例化对象，如何引用和废除对象，以及绑定的概念。

ECMAScript 对象类型

本节介绍了 ECMAScript 的三种类型：本地对象、内置对象和宿主对象，并提供了指向相关参考手册的链接。

ECMAScript 对象作用域

本节讲解了 ECMAScript 作用域以及 this 关键字。

ECMAScript 定义类或对象

本节详细讲解了创建 ECMAScript 对象或类的各种方式。

ECMAScript 修改对象

本节讲解了如何通过创建新方法或重定义已有方法来修改对象。

JavaScript Number 对象

JavaScript 只有一种数字类型。

可以使用也可以不使用小数点来书写数字。

JavaScript 数字

JavaScript 数字可以使用也可以不使用小数点来书写：

实例

```
var pi=3.14;    // 使用小数点
var x=34;       // 不使用小数点
```

极大或极小的数字可通过科学（指数）计数法来写：

实例

```
var y=123e5;    // 12300000
var z=123e-5;   // 0.00123
```

所有 JavaScript 数字均为 64 位

JavaScript 不是类型语言。与许多其他编程语言不同，JavaScript 不定义不同类型的数字，比如整数、短、长、浮点等等。

JavaScript 中的所有数字都存储为根为 10 的 64 位（8 比特），浮点数。

精度

整数（不使用小数点或指数计数法）最多为 15 位。

小数的最大位数是 17，但是浮点运算并不总是 100% 准确：

实例

```
var x=0.2+0.1;
```

八进制和十六进制

如果前缀为 0，则 JavaScript 会把数值常量解释为八进制数，如果前缀为 0 和 "x"，则解释为十六进制数。

实例

```
var y=0377;
var z=0xFF;
```

提示：绝不要在数字前面写零，除非您需要进行八进制转换。

数字属性和方法

属性：

- MAX VALUE
- MIN VALUE
- NEGATIVE INFINITIVE
- POSITIVE INFINITIVE
- NaN
- prototype
- constructor

方法：

- toExponential()
- toFixed()
- toPrecision()
- toString()
- valueOf()

完整的 Number 对象参考手册

如需可用于 Number 对象的所有属性和方法的完整参考，请访问我们的 Number 对象参考手册。

该参考手册包含每个属性和方法的描述和实例。

JavaScript 字符串(String)对象

String 对象用于处理已有的字符块。

JavaScript String（字符串）对象 实例

计算字符串的长度

如何使用长度属性来计算字符串的长度。

```
<html>
<body>

<script type="text/javascript">

var txt="Hello World!"
document.write(txt.length)

</script>

</body>
</html>
```

为字符串添加样式

如何为字符串添加样式。

```
<html>
<body>

<script type="text/javascript">

var txt="Hello World!"

document.write("<p>Big: " + txt.big() + "</p>")
document.write("<p>Small: " + txt.small() + "</p>")

document.write("<p>Bold: " + txt.bold() + "</p>")
document.write("<p>Italic: " + txt.italics() + "</p>")

document.write("<p>Blink: " + txt.blink() + " (does not work in IE)</p>")
document.write("<p>Fixed: " + txt.fixed() + "</p>")
document.write("<p>Strike: " + txt.strike() + "</p>")

document.write("<p>Fontcolor: " + txt.fontcolor("Red") + "</p>")
document.write("<p>Fontsize: " + txt.fontsize(16) + "</p>")

document.write("<p>Lowercase: " + txt.toLowerCase() + "</p>")
document.write("<p>Uppercase: " + txt.toUpperCase() + "</p>")

document.write("<p>Subscript: " + txt.sub() + "</p>")
document.write("<p>Superscript: " + txt.sup() + "</p>")

document.write("<p>Link: " + txt.link("http://www.w3school.com.cn") + "</p>")
</script>

</body>
</html>
```

indexOf() 方法

如何使用 indexOf() 来定位字符串中某一个指定的字符首次出现的位置。

```
<html>
<body>

<script type="text/javascript">

var str="Hello world!"
document.write(str.indexOf("Hello") + "<br />")
document.write(str.indexOf("World") + "<br />")
document.write(str.indexOf("world"))

</script>

</body>
</html>
```

match() 方法

如何使用 match() 来查找字符串中特定的字符，并且如果找到的话，则返回这个字符。

```
<html>
<body>

<script type="text/javascript">

var str="Hello world!"
document.write(str.match("world") + "<br />")
document.write(str.match("World") + "<br />")
document.write(str.match("worlld") + "<br />")
document.write(str.match("world!"))

</script>

</body>
</html>
```

如何替换字符串中的字符 - replace()

如何使用 replace() 方法在字符串中用某些字符替换另一些字符。

```
<html>
<body>

<script type="text/javascript">

var str="Visit Microsoft!"
document.write(str.replace(/Microsoft/,"W3School"))

</script>
</body>
</html>
```

完整的 String 对象参考手册

请查看我们的 JavaScript String 对象参考手册，其中提供了可以与字符串对象一同使用的所有的属性和方法。

这个手册包含的关于每个属性和方法的用法的详细描述和实例。

字符串对象

字符串对象用于处理已有的字符块。

例子：

下面的例子使用字符串对象的长度属性来计算字符串的长度。

```
var txt="Hello world!"
document.write(txt.`length`)
```

上面的代码输出为：

```
12
```

下面的例子使用字符串对象的toUpperCase()方法将字符串转换为大写：

```
var txt="Hello world!"
document.write(txt.`toUpperCase()`)
```

上面的代码输出为：

```
HELLO WORLD!
```

相关页面

JavaScript 高级教程：ECMAScript 类型转换

JavaScript 高级教程：ECMAScript 引用类型

JavaScript 参考手册：String 对象

JavaScript Date（日期）对象

日期对象用于处理日期和时间。

JavaScript Date（日期）对象 实例

返回当日的日期和时间

如何使用 Date() 方法获得当日的日期。

```
<html>
<body>

<script type="text/javascript">

document.write(Date())

</script>

</body>
</html>
```

getTime()

getTime() 返回从 1970 年 1 月 1 日至今的毫秒数。

```
<html>
<body>

<script type="text/javascript">
var d=new Date();
document.write("从 1970/01/01 至今已过去 " + d.getTime() + " 毫秒");
</script>

</body>
</html>
```

setFullYear()

如何使用 setFullYear() 设置具体的日期。

```
<html>
<body>

<script type="text/javascript">

var d = new Date()
d.setFullYear(1992,10,3)
document.write(d)

</script>

</body>
</html>
```

toUTCString()

如何使用 toUTCString() 将当日的日期（根据 UTC）转换为字符串。

```
<html>
<body>

<script type="text/javascript">

var d = new Date()
document.write (d.toUTCString())

</script>

</body>
</html>
```

getDay()

如何使用 getDay() 和数组来显示星期，而不仅仅是数字。

```
<html>
<body>

<script type="text/javascript">

var d=new Date()
var weekday=new Array(7)
weekday[0]="星期日"
weekday[1]="星期一"
weekday[2]="星期二"
weekday[3]="星期三"
weekday[4]="星期四"
weekday[5]="星期五"
weekday[6]="星期六"

document.write("今天是" + weekday[d.getDay()])

</script>

</body>
</html>
```

显示一个钟表

如何在网页上显示一个钟表。

```
<html>
<head>
<script type="text/javascript">
function startTime()
{
var today=new Date()
var h=today.getHours()
var m=today.getMinutes()
var s=today.getSeconds()
// add a zero in front of numbers<10
m=checkTime(m)
s=checkTime(s)
document.getElementById('txt').innerHTML=h+":"+m+":"+s
t=setTimeout('startTime()',500)
}

function checkTime(i)
{
if (i<10)
  {i="0" + i}
  return i
}
</script>
</head>

<body onload="startTime()">
<div id="txt"></div>
</body>
</html>
```

完整的 Date 对象参考手册

我们提供 JavaScript Date 对象参考手册，其中包括所有可用于日期对象的属性和方法。

该手册包含了对每个属性和方法的详细描述以及相关实例。

定义日期

Date 对象用于处理日期和时间。

可以通过 new 关键词来定义 Date 对象。以下代码定义了名为 myDate 的 Date 对象：

```
var myDate=`new` Date()
```

注释：Date 对象自动使用当前的日期和时间作为其初始值。

操作日期

通过使用针对日期对象的方法，我们可以很容易地对日期进行操作。

在下面的例子中，我们为日期对象设置了一个特定的日期 (2008 年 8 月 9 日)：

```
var myDate=new Date()
myDate.`setFullYear`(2008,7,9)
```

注意：表示月份的参数介于 0 到 11 之间。也就是说，如果希望把月设置为 8 月，则参数应该是 7。

在下面的例子中，我们将日期对象设置为 5 天后的日期：

```
var myDate=new Date()
myDate.`setDate`(myDate.getDate()+5)
```

注意：如果增加天数会改变月份或者年份，那么日期对象会自动完成这种转换。

比较日期

日期对象也可用于比较两个日期。

下面的代码将当前日期与 2008 年 8 月 9 日做了比较：

```
var myDate=new Date();
myDate.setFullYear(2008,8,9);

var today = new Date();

if (myDate>today)
{
alert("Today is before 9th August 2008");
}
else
{
alert("Today is after 9th August 2008");
}
```

JavaScript Array（数组）对象

数组对象的作用是：使用单独的变量名来存储一系列的值。

实例

创建数组

创建数组，为其赋值，然后输出这些值。

```
<html>
<body>

<script type="text/javascript">
var mycars = new Array()
mycars[0] = "Saab"
mycars[1] = "Volvo"
mycars[2] = "BMW"

for (i=0;i<mycars.length;i++)
{
document.write(mycars[i] + "<br />")
}
</script>

</body>
</html>
```

For...In 声明

使用 for...in 声明来循环输出数组中的元素。

```
<html>
<body>
<script type="text/javascript">
var x
var mycars = new Array()
mycars[0] = "Saab"
mycars[1] = "Volvo"
mycars[2] = "BMW"

for (x in mycars)
{
document.write(mycars[x] + "<br />")
}
</script>
</body>
</html>
```

合并两个数组 - concat()

如何使用 concat() 方法来合并两个数组。

```
<html>
<body>

<script type="text/javascript">

var arr = new Array(3)
arr[0] = "George"
arr[1] = "John"
arr[2] = "Thomas"

var arr2 = new Array(3)
arr2[0] = "James"
arr2[1] = "Adrew"
arr2[2] = "Martin"

document.write(arr.concat(arr2))

</script>

</body>
</html>
```

用数组的元素组成字符串 - join()

如何使用 join() 方法将数组的所有元素组成一个字符串。

```
<html>
<body>

<script type="text/javascript">

var arr = new Array(3);
arr[0] = "George"
arr[1] = "John"
arr[2] = "Thomas"

document.write(arr.join());

document.write("<br />");

document.write(arr.join("."));

</script>

</body>
</html>
```

文字数组 - sort()

如何使用 sort() 方法从字面上对数组进行排序。

```
<html>
<body>

<script type="text/javascript">

var arr = new Array(6)
arr[0] = "George"
arr[1] = "John"
arr[2] = "Thomas"
arr[3] = "James"
arr[4] = "Adrew"
arr[5] = "Martin"

document.write(arr + "<br />")
document.write(arr.sort())

</script>

</body>
</html>
```

数字数组 - sort()

如何使用 sort() 方法从数值上对数组进行排序。

```
<html>
<body>

<script type="text/javascript">

function sortNumber(a, b)
{
return a - b
}

var arr = new Array(6)
arr[0] = "10"
arr[1] = "5"
arr[2] = "40"
arr[3] = "25"
arr[4] = "1000"
arr[5] = "1"

document.write(arr + "<br />")
document.write(arr.sort(sortNumber))

</script>

</body>
</html>
```

完整的 Array 对象参考手册

我们提供 JavaScript Array 对象参考手册，其中包括所有可用于数组对象的属性和方法。

该手册包含了对每个属性和方法的详细描述以及相关实例。

定义数组

数组对象用来在单独的变量名中存储一系列的值。

我们使用关键词 new 来创建数组对象。下面的代码定义了一个名为 myArray 的数组对象：

```
var myArray=new Array()
```

有两种向数组赋值的方法（你可以添加任意多的值，就像你可以定义你需要的任意多的变量一样）。

1:

```
var mycars=`new Array()`
mycars[0]="Saab"
mycars[1]="Volvo"
mycars[2]="BMW"
```

也可以使用一个整数自变量来控制数组的容量：

```
var mycars=`new Array(3)`
mycars[0]="Saab"
mycars[1]="Volvo"
mycars[2]="BMW"
```

2:

```
var mycars=`new Array("Saab","Volvo","BMW")`
```

注意：如果你需要在数组内指定数值或者逻辑值，那么变量类型应该是数值变量或者布尔变量，而不是字符变量。

访问数组

通过指定数组名以及索引号码，你可以访问某个特定的元素。

下面是代码行：

```
document.write(mycars[0])
```

下面是输出：

```
Saab
```

修改已有数组中的值

如需修改已有数组中的值，只要向指定下标号添加一个新值即可：

```
mycars[0]="Opel";
```

现在，以上代码：

```
document.write(mycars[0]);
```

将输出：

```
Opel
```

JavaScript Boolean（逻辑）对象

Boolean（逻辑）对象用于将非逻辑值转换为逻辑值（**true** 或者 **false**）。

实例

检查逻辑值

检查逻辑对象是 true 还是 false。

```
<html>
<body>

<script type="text/javascript">
var b1=new Boolean( 0)
var b2=new Boolean(1)
var b3=new Boolean("")
var b4=new Boolean(null)
var b5=new Boolean(NaN)
var b6=new Boolean("false")

document.write("0 是逻辑的 "+ b1 +"<br />")
document.write("1 是逻辑的 "+ b2 +"<br />")
document.write("空字符串是逻辑的 "+ b3 + "<br />")
document.write("null 是逻辑的 "+ b4+ "<br />")
document.write("NaN 是逻辑的 "+ b5 +"<br />")
document.write("字符串 'false' 是逻辑的 "+ b6 +"<br />")
</script>

</body>
</html>
```

完整的 Boolean 对象参考手册

我们提供 JavaScript Boolean 对象参考手册，其中包括所有可用于逻辑对象的属性和方法。

该手册包含了对每个属性和方法的详细描述以及相关实例。

Boolean 对象

您可以将 Boolean 对象理解为一个产生逻辑值的对象包装器。

Boolean（逻辑）对象用于将非逻辑值转换为逻辑值（true 或者 false）。

创建 Boolean 对象

使用关键词 new 来定义 Boolean 对象。下面的代码定义了一个名为 myBoolean 的逻辑对象：

```
var myBoolean=new Boolean()
```

注释：如果逻辑对象无初始值或者其值为 0、-0、null、""、false、undefined 或者 NaN，那么对象的值为 false。否则，其值为 true（即使当自变量为字符串 "false" 时）！

下面的所有的代码行均会创建初始值为 false 的 Boolean 对象。

```
var myBoolean=new Boolean();
var myBoolean=new Boolean(0);
var myBoolean=new Boolean(null);
var myBoolean=new Boolean("");
var myBoolean=new Boolean(false);
var myBoolean=new Boolean(NaN);
```

下面的所有的代码行均会创初始值为 true 的 Boolean 对象：

```
var myBoolean=new Boolean(1);
var myBoolean=new Boolean(true);
var myBoolean=new Boolean("true");
var myBoolean=new Boolean("false");
var myBoolean=new Boolean("Bill Gates");
```

相关页面

JavaScript 高级教程：ECMAScript 引用类型

JavaScript 参考手册：Boolean 对象

JavaScript Math（算数）对象

Math（算数）对象的作用是：执行常见的算数任务。

实例

round()

如何使用 round()。

```
<html>
<body>

<script type="text/javascript">

document.write(Math.round(0.60) + "<br />")
document.write(Math.round(0.50) + "<br />")
document.write(Math.round(0.49) + "<br />")
document.write(Math.round(-4.40) + "<br />")
document.write(Math.round(-4.60))

</script>

</body>
</html>
```

random()

如何使用 random() 来返回 0 到 1 之间的随机数。

```
<html>
<body>

<script type="text/javascript">

document.write(Math.random())

</script>

</body>
</html>
```

max()

如何使用 max() 来返回两个给定的数中的较大的数。（在 ECMASCript v3 之前，该方法只有两个参数。）

```
<html>
<body>

<script type="text/javascript">

document.write(Math.max(5,7) + "<br />")
document.write(Math.max(-3,5) + "<br />")
document.write(Math.max(-3,-5) + "<br />")
document.write(Math.max(7.25,7.30))

</script>

</body>
</html>
```

min()

如何使用 min() 来返回两个给定的数中的较小的数。（在 ECMASCript v3 之前，该方法只有两个参数。）

```
<html>
<body>

<script type="text/javascript">

document.write(Math.min(5,7) + "<br />")
document.write(Math.min(-3,5) + "<br />")
document.write(Math.min(-3,-5) + "<br />")
document.write(Math.min(7.25,7.30))

</script>

</body>
</html>
```

完整的 Math 对象参考手册

我们提供 JavaScript Math 对象的参考手册，其中包括所有可用于算术对象的属性和方法。

该手册包含了对每个属性和方法的详细描述以及相关实例。

Math 对象

Math（算数）对象的作用是：执行普通的算数任务。

Math 对象提供多种算数值类型和函数。无需在使用这个对象之前对它进行定义。

算数值

JavaScript 提供 8 种可被 Math 对象访问的算数值：

- 常数

- 圆周率
- 2 的平方根
- 1/2 的平方根
- 2 的自然对数
- 10 的自然对数
- 以 2 为底的 e 的对数
- 以 10 为底的 e 的对数

这是在 Javascript 中使用这些值的方法：（与上面的算数值一一对应）

- Math.E
- Math.PI
- Math.SQRT2
- Math.SQRT1_2
- Math.LN2
- Math.LN10
- Math.LOG2E
- Math.LOG10E

算数方法

除了可被 Math 对象访问的算数值以外，还有几个函数（方法）可以使用。

函数（方法）实例：

下面的例子使用了 Math 对象的 round 方法对一个数进行四舍五入。

```
document.write(`Math.round(4.7)`)
```

上面的代码输出为：

```
5
```

下面的例子使用了 Math 对象的 random() 方法来返回一个介于 0 和 1 之间的随机数：

```
document.write(`Math.random()`)
```

上面的代码输出为：

```
0.9370844220218102
```

下面的例子使用了 Math 对象的 floor() 方法和 random() 来返回一个介于 0 和 10 之间的随机数：

```
document.write(`Math.floor(Math.random()*11)`)
```

上面的代码输出为：

```
3
```

JavaScript RegExp 对象

RegExp 对象用于规定在文本中检索的内容。

什么是 RegExp？

RegExp 是正则表达式的缩写。

当您检索某个文本时，可以使用一种模式来描述要检索的内容。RegExp 就是这种模式。

简单的模式可以是一个单独的字符。

更复杂的模式包括了更多的字符，并可用于解析、格式检查、替换等等。

您可以规定字符串中的检索位置，以及要检索的字符类型，等等。

定义 RegExp

RegExp 对象用于存储检索模式。

通过 new 关键词来定义 RegExp 对象。以下代码定义了名为 patt1 的 RegExp 对象，其模式是 "e"：

```
var patt1=new RegExp("e");
```

当您使用该 RegExp 对象在一个字符串中检索时，将寻找的是字符 "e"。

RegExp 对象的方法

RegExp 对象有 3 个方法：test()、exec() 以及 compile()。

test()

test() 方法检索字符串中的指定值。返回值是 true 或 false。

例子：

```
var patt1=new RegExp("e");

document.write(patt1.test("The best things in life are free"));
```

由于该字符串中存在字母 "e"，以上代码的输出将是：

```
true
```

TIY

exec()

exec() 方法检索字符串中的指定值。返回值是被找到的值。如果没有发现匹配，则返回 null。

例子 1：

```
var patt1=new RegExp("e");

document.write(patt1.exec("The best things in life are free"));
```

由于该字符串中存在字母 "e"，以上代码的输出将是：

```
e
```

TIY

例子 2：

您可以向 RegExp 对象添加第二个参数，以设定检索。例如，如果需要找到所有某个字符的所有存在，则可以使用 "g" 参数 ("global")。

如需关于如何修改搜索模式的完整信息，请访问我们的 RegExp 对象参考手册。

在使用 "g" 参数时，exec() 的工作原理如下：

- 找到第一个 "e"，并存储其位置
- 如果再次运行 exec()，则从存储的位置开始检索，并找到下一个 "e"，并存储其位置

```
var patt1=new RegExp("e","g");
do
{
result=patt1.exec("The best things in life are free");
document.write(result);
}
while (result!=null)
```

由于这个字符串中 6 个 "e" 字母，代码的输出将是：

```
eeeeeenull
```

TIY

compile()

compile() 方法用于改变 RegExp。

compile() 既可以改变检索模式，也可以添加或删除第二个参数。

例子：

```
var patt1=new RegExp("e");

document.write(patt1.test("The best things in life are free"));

patt1.compile("d");

document.write(patt1.test("The best things in life are free"));
```

由于字符串中存在 "e"，而没有 "d"，以上代码的输出是：

```
truefalse
```

TIY

完整的 **RegExp** 对象参考手册

如需可与 RegExp 对象一同使用所有属性和方法的完整信息，请访问我们的 RegExp 对象参考手册。

这个参考手册包含了对 RegExp 对象中每个属性和方法的详细描述，以及使用的例子。

JavaScript BOM

JavaScript Window - 浏览器对象模型

浏览器对象模型 **(BOM)** 使 **JavaScript** 有能力与浏览器“对话”。

浏览器对象模型 (BOM)

浏览器对象模型（Browser Object Model）尚无正式标准。

由于现代浏览器已经（几乎）实现了 JavaScript 交互性方面的相同方法和属性，因此常被认为是 BOM 的方法和属性。

Window 对象

所有浏览器都支持 *window* 对象。它表示浏览器窗口。

所有 JavaScript 全局对象、函数以及变量均自动成为 window 对象的成员。

全局变量是 window 对象的属性。

全局函数是 window 对象的方法。

甚至 HTML DOM 的 document 也是 window 对象的属性之一：

```
window.document.getElementById("header");
```

与此相同：

```
document.getElementById("header");
```

Window 尺寸

有三种方法能够确定浏览器窗口的尺寸（浏览器的视口，不包括工具栏和滚动条）。

对于Internet Explorer、Chrome、Firefox、Opera 以及 Safari：

- window.innerHeight - 浏览器窗口的内部高度
- window.innerWidth - 浏览器窗口的内部宽度

对于 Internet Explorer 8、7、6、5：

- document.documentElement.clientHeight

- document.documentElement.clientWidth

或者

- document.body.clientHeight
- document.body.clientWidth

实用的 JavaScript 方案（涵盖所有浏览器）：

实例

```
var w=window.innerWidth
|| document.documentElement.clientWidth
|| document.body.clientWidth;

var h=window.innerHeight
|| document.documentElement.clientHeight
|| document.body.clientHeight;
```

该例显示浏览器窗口的高度和宽度：（不包括工具栏/滚动条）

其他 Window 方法

一些其他方法：

- window.open() - 打开新窗口
- window.close() - 关闭当前窗口
- window.moveTo() - 移动当前窗口
- window.resizeTo() - 调整当前窗口的尺寸

JavaScript Window Screen

window.screen 对象包含有关用户屏幕的信息。

Window Screen

window.screen 对象在编写时可以不使用 window 这个前缀。

一些属性：

- screen.availWidth - 可用的屏幕宽度
- screen.availHeight - 可用的屏幕高度

Window Screen 可用宽度

screen.availWidth 属性返回访问者屏幕的宽度，以像素计，减去界面特性，比如窗口任务栏。

实例

返回您的屏幕的可用宽度：

```
<script>
document.write("可用宽度：" + `screen.availWidth`);
</script>
```

以上代码输出为：

```
<script>document.write("可用宽度：" + screen.availWidth);</script>可用宽度：1366
```

Window Screen 可用高度

screen.availHeight 属性返回访问者屏幕的高度，以像素计，减去界面特性，比如窗口任务栏。

实例

返回您的屏幕的可用高度：

```
<script>

document.write("可用高度：" + `screen.availHeight`);

</script>
```

以上代码输出为：

```
<script>document.write("可用高度：" + screen.availHeight);</script>可用高度：768
```

JavaScript Window Location

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。

Window Location

window.location 对象在编写时可不使用 window 这个前缀。

一些例子：

- location.hostname 返回 web 主机的域名
- location.pathname 返回当前页面的路径和文件名
- location.port 返回 web 主机的端口 （80 或 443）
- location.protocol 返回所使用的 web 协议（http:// 或 https://）

Window Location Href

location.href 属性返回当前页面的 URL。

实例

返回（当前页面的）整个 URL：

```
<script>
document.write(`location.href`);
</script>
```

以上代码输出为：

```
<script>document.write(location.href);</script>http://www.w3school.com.cn/js/js_window_lo
```

Window Location Pathname

location.pathname 属性返回 URL 的路径名。

实例

返回当前 URL 的路径名：

```
<script>

document.write(`location.pathname`);

</script>
```

以上代码输出为：

```
<script>document.write(location.pathname);</script>/js/js_window_location.asp
```

Window Location Assign

location.assign() 方法加载新的文档。

实例

加载一个新的文档：

```
<html>
<head>
<script>
function newDoc()
  {
  `window.location.assign("http://www.w3school.com.cn")`
  }
</script>
</head>
<body>

<input type="button" value="加载新文档" onclick="newDoc()">

</body>
</html>
```

JavaScript Window History

window.history 对象包含浏览器的历史。

Window History

window.history 对象在编写时可不使用 window 这个前缀。

为了保护用户隐私，对 JavaScript 访问该对象的方法做出了限制。

一些方法：

- history.back() - 与在浏览器点击后退按钮相同
- history.forward() - 与在浏览器中点击按钮向前相同

Window History Back

history.back() 方法加载历史列表中的前一个 URL。

这与在浏览器中点击后退按钮是相同的：

实例

在页面上创建后退按钮：

```
<html>
<head>
<script>
function goBack()
  {
  `window.history.back()`
  }
</script>
</head>
<body>

<input type="button" value="Back" onclick="goBack()">

</body>
</html>
```

以上代码输出为：

```
<input type="button" value="转到上一页" onclick="goBack()">
```

Window History Forward

history forward() 方法加载历史列表中的下一个 URL。

这与在浏览器中点击前进按钮是相同的：

实例

在页面上创建一个向前的按钮：

```
<html>
<head>
<script>
function goForward()
  {
  `window.history.forward()`
  }
</script>
</head>
<body>

<input type="button" value="Forward" onclick="goForward()">

</body>
</html>
```

以上代码输出为：

```
<input type="button" value="转到下一页" onclick="goForward()">
```

JavaScript Window Navigator

window.navigator 对象包含有关访问者浏览器的信息。

Window Navigator

window.navigator 对象在编写时可不使用 window 这个前缀。

实例

```
<div id="example"></div>

<script>

txt = "<p>Browser CodeName: " + navigator.appCodeName + "</p>";
txt+= "<p>Browser Name: " + navigator.appName + "</p>";
txt+= "<p>Browser Version: " + navigator.appVersion + "</p>";
txt+= "<p>Cookies Enabled: " + navigator.cookieEnabled + "</p>";
txt+= "<p>Platform: " + navigator.platform + "</p>";
txt+= "<p>User-agent header: " + navigator.userAgent + "</p>";
txt+= "<p>User-agent language: " + navigator.systemLanguage + "</p>";

document.getElementById("example").innerHTML=txt;

</script>
```

警告：来自 navigator 对象的信息具有误导性，不应该被用于检测浏览器版本，这是因为：

- navigator 数据可被浏览器使用者更改
- 浏览器无法报告晚于浏览器发布的新操作系统

浏览器检测

由于 navigator 可误导浏览器检测，使用对象检测可用来嗅探不同的浏览器。

由于不同的浏览器支持不同的对象，您可以使用对象来检测浏览器。例如，由于只有 Opera 支持属性 "window.opera"，您可以据此识别出 Opera。

例子：if (window.opera) {...some action...}

JavaScript 消息框

可以在 **JavaScript** 中创建三种消息框：警告框、确认框、提示框。

实例

警告框

```
<html>
<head>
<script type="text/javascript">
function disp_alert()
{
alert("我是警告框！！")
}
</script>
</head>
<body>

<input type="button" onclick="disp_alert()" value="显示警告框" />

</body>
</html>
```

带有折行的警告框

```
<html>
<head>
<script type="text/javascript">
function disp_alert()
{
alert("再次向您问好！在这里，我们向您演示" + '\n' + "如何向警告框添加折行。")
}
</script>
</head>
<body>

<input type="button" onclick="disp_alert()" value="显示警告框" />

</body>
</html>
```

确认框

```
<html>
<head>
<script type="text/javascript">
function show_confirm()
{
var r=confirm("Press a button!");
if (r==true)
  {
  alert("You pressed OK!");
  }
else
  {
  alert("You pressed Cancel!");
  }
}
</script>
</head>
<body>

<input type="button" onclick="show_confirm()" value="Show a confirm box" />

</body>
</html>
```

提示框

```
<html>
<head>
<script type="text/javascript">
function disp_prompt()
  {
  var name=prompt("请输入您的名字","Bill Gates")
  if (name!=null && name!="")
    {
    document.write("你好！" + name + " 今天过得怎么样？")
    }
  }
</script>
</head>
<body>

<input type="button" onclick="disp_prompt()" value="显示提示框" />

</body>
</html>
```

警告框

警告框经常用于确保用户可以得到某些信息。

当警告框出现后，用户需要点击确定按钮才能继续进行操作。

语法：

```
alert("文本")
```

确认框

确认框用于使用户可以验证或者接受某些信息。

当确认框出现后，用户需要点击确定或者取消按钮才能继续进行操作。

如果用户点击确认，那么返回值为 true。如果用户点击取消，那么返回值为 false。

语法：

```
confirm("文本")
```

提示框

提示框经常用于提示用户在进入页面前输入某个值。

当提示框出现后，用户需要输入某个值，然后点击确认或取消按钮才能继续操纵。

如果用户点击确认，那么返回值为输入的值。如果用户点击取消，那么返回值为 null。

语法：

```
prompt("文本","默认值")
```

JavaScript 计时

通过使用 **JavaScript**，我们有能力做到在一个设定的时间间隔之后来执行代码，而不是在函数被调用后立即执行。我们称之为计时事件。

实例

简单的计时

单击本例中的按钮后，会在 5 秒后弹出一个警告框。

```
<html>
<head>
<script type="text/javascript">
function timedMsg()
{
var t=setTimeout("alert('5 秒！')",5000)
}
</script>
</head>

<body>
<form>
<input type="button" value="显示定时的警告框" onClick = "timedMsg()">
</form>
<p>请点击上面的按钮。警告框会在 5 秒后显示。</p>
</body>

</html>
```

另一个简单的计时

本例中的程序会执行 2 秒、4 秒和 6 秒的计时。

```
<html>
<head>
<script type="text/javascript">
function timedText()
{
var t1=setTimeout("document.getElementById('txt').value='2 秒'",2000)
var t2=setTimeout("document.getElementById('txt').value='4 秒'",4000)
var t3=setTimeout("document.getElementById('txt').value='6 秒'",6000)
}
</script>
</head>

<body>

<form>
<input type="button" value="显示计时的文本" onClick="timedText()">
<input type="text" id="txt">
</form>

<p>点击上面的按钮。输入框会显示出已经逝去的时间（2、4、6 秒）。</p>
</body>

</html>
```

在一个无穷循环中的计时事件

在本例中，单击开始计时按钮后，程序开始从 0 以秒计时。

```
<html>
<head>
<script type="text/javascript">
var c=0
var t
function timedCount()
{
document.getElementById('txt').value=c
c=c+1
t=setTimeout("timedCount()",1000)
}
</script>
</head>

<body>

<form>
<input type="button" value="开始计时！" onClick="timedCount()">
<input type="text" id="txt">
</form>

<p>请点击上面的按钮。输入框会从 0 开始一直进行计时。</p>

</body>

</html>
```

带有停止按钮的无穷循环中的计时事件

在本例中，点击计数按钮后根据用户输入的数值开始倒计时，点击停止按钮停止计时。

```
<html>
<head>
<script type="text/javascript">
var c=0
var t
function timedCount()
{
document.getElementById('txt').value=c
c=c+1
t=setTimeout("timedCount()",1000)
}

function stopCount()
{
c=0;
setTimeout("document.getElementById('txt').value=0",0);
clearTimeout(t);
}
</script>
</head>

<body>

<form>
<input type="button" value="开始计时！" onClick="timedCount()">
<input type="text" id="txt">
<input type="button" value="停止计时！" onClick="stopCount()">
</form>

<p>请点击上面的“开始计时”按钮来启动计时器。输入框会一直进行计时，从 0 开始。点击“停止计时”按钮可以终止计

</body>

</html>
```

使用计时事件制作的钟表

一个 JavaScript 小时钟

```
<html>
<head>
<script type="text/javascript">
function startTime()
{
var today=new Date()
var h=today.getHours()
var m=today.getMinutes()
var s=today.getSeconds()
// add a zero in front of numbers<10
m=checkTime(m)
s=checkTime(s)
document.getElementById('txt').innerHTML=h+":"+m+":"+s
t=setTimeout('startTime()',500)
}

function checkTime(i)
{
if (i<10)
  {i="0" + i}
  return i
}
</script>
</head>

<body onload="startTime()">
<div id="txt"></div>
</body>
</html>
```

JavaScript 计时事件

通过使用 JavaScript，我们有能力作到在一个设定的时间间隔之后来执行代码，而不是在函数被调用后立即执行。我们称之为计时事件。

在 JavaScritp 中使用计时事件是很容易的，两个关键方法是:

setTimeout()

未来的某时执行代码

clearTimeout()

取消setTimeout()

setTimeout()

语法

```
var t=setTimeout("javascript语句",毫秒)
```

setTimeout() 方法会返回某个值。在上面的语句中，值被储存在名为 t 的变量中。假如你希望取消这个 setTimeout()，你可以使用这个变量名来指定它。

setTimeout() 的第一个参数是含有 JavaScript 语句的字符串。这个语句可能诸如 "alert('5 seconds!')"，或者对函数的调用，诸如 alertMsg()"。

第二个参数指示从当前起多少毫秒后执行第一个参数。

提示：1000 毫秒等于一秒。

实例

当下面这个例子中的按钮被点击时，一个提示框会在5秒中后弹出。

```
<html>
<head>
<script type="text/javascript">
function timedMsg()
 {
 `var t=setTimeout("alert('5 seconds!')",5000)`
 }
</script>
</head>

<body>
<form>
<input type="button" value="Display timed alertbox!" onClick="timedMsg()">
</form>
</body>
</html>
```

实例 - 无穷循环

要创建一个运行于无穷循环中的计时器，我们需要编写一个函数来调用其自身。在下面的例子中，当按钮被点击后，输入域便从 0 开始计数。

```
<html>

<head>
<script type="text/javascript">
var c=0
var t
function timedCount()
 {
 document.getElementById('txt').value=c
 c=c+1
 `t=setTimeout("timedCount()",1000)`
 }
</script>
</head>

<body>
<form>
<input type="button" value="Start count!" onClick="timedCount()">
<input type="text" id="txt">
</form>
</body>

</html>
```

clearTimeout()

语法

```
clearTimeout(setTimeout_variable)
```

实例

下面的例子和上面的无穷循环的例子相似。唯一的不同是，现在我们添加了一个 "Stop Count!" 按钮来停止这个计数器：

```
<html>

<head>
<script type="text/javascript">
var c=0
var t

function timedCount()
 {
 document.getElementById('txt').value=c
 c=c+1
 `t=setTimeout("timedCount()",1000)`
 }

function stopCount()
 {
 `clearTimeout(t)`
 }
</script>
</head>

<body>
<form>
<input type="button" value="Start count!" onClick="timedCount()">
<input type="text" id="txt">
<input type="button" value="Stop count!" onClick="stopCount()">
</form>
</body>

</html>
```

JavaScript Cookies

cookie 用来识别用户。

实例

创建一个欢迎 cookie

利用用户在提示框中输入的数据创建一个 JavaScript Cookie，当该用户再次访问该页面时，根据 cookie 中的信息发出欢迎信息。

```
<html>
<head>
<script type="text/javascript">
function getCookie(c_name)
{
if (document.cookie.length>0)
{
c_start=document.cookie.indexOf(c_name + "=")
if (c_start!=-1)
{
c_start=c_start + c_name.length+1
c_end=document.cookie.indexOf(";",c_start)
if (c_end==-1) c_end=document.cookie.length
return unescape(document.cookie.substring(c_start,c_end))
}
}
return ""
}

function setCookie(c_name,value,expiredays)
{
var exdate=new Date()
exdate.setDate(exdate.getDate()+expiredays)
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : "; expires="+exdate.toGMTString())
}

function checkCookie()
{
username=getCookie('username')
if (username!=null && username!="")
  {alert('Welcome again '+username+'!')}
else
  {
  username=prompt('Please enter your name:',"")
  if (username!=null && username!="")
    {
    setCookie('username',username,365)
    }
  }
}
</script>
</head>
<body onLoad="checkCookie()">
</body>
</html>
```

什么是cookie?

cookie 是存储于访问者的计算机中的变量。每当同一台计算机通过浏览器请求某个页面时，就会发送这个 cookie。你可以使用 JavaScript 来创建和取回 cookie 的值。

有关cookie的例子：

名字 cookie

当访问者首次访问页面时，他或她也许会填写他/她们的名字。名字会存储于 cookie 中。当访问者再次访问网站时，他们会收到类似 "Welcome John Doe!" 的欢迎词。而名字则是从 cookie 中取回的。

密码 cookie

当访问者首次访问页面时，他或她也许会填写他/她们的密码。密码也可被存储于 cookie 中。当他们再次访问网站时，密码就会从 cookie 中取回。

日期 cookie

当访问者首次访问你的网站时，当前的日期可存储于 cookie 中。当他们再次访问网站时，他们会收到类似这样的一条消息："Your last visit was on Tuesday August 11, 2005!"。日期也是从 cookie 中取回的。

创建和存储 cookie

在这个例子中我们要创建一个存储访问者名字的 cookie。当访问者首次访问网站时，他们会被要求填写姓名。名字会存储于 cookie 中。当访问者再次访问网站时，他们就会收到欢迎词。

首先，我们会创建一个可在 cookie 变量中存储访问者姓名的函数：

```
function setCookie(c_name,value,expiredays)
{
var exdate=new Date()
exdate.setDate(exdate.getDate()+expiredays)
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : ";expires="+exdate.toGMTString())
}
```

上面这个函数中的参数存有 cookie 的名称、值以及过期天数。

在上面的函数中，我们首先将天数转换为有效的日期，然后，我们将 cookie 名称、值及其过期日期存入 document.cookie 对象。

之后，我们要创建另一个函数来检查是否已设置 cookie：

```
function getCookie(c_name)
{
if (document.cookie.length>0)
  {
  c_start=document.cookie.indexOf(c_name + "=")
  if (c_start!=-1)
    {
    c_start=c_start + c_name.length+1
    c_end=document.cookie.indexOf(";",c_start)
    if (c_end==-1) c_end=document.cookie.length
    return unescape(document.cookie.substring(c_start,c_end))
    }
  }
return ""
}
```

上面的函数首先会检查 document.cookie 对象中是否存有 cookie。假如 document.cookie 对象存有某些 cookie，那么会继续检查我们指定的 cookie 是否已储存。如果找到了我们要的 cookie，就返回值，否则返回空字符串。

最后，我们要创建一个函数，这个函数的作用是：如果 cookie 已设置，则显示欢迎词，否则显示提示框来要求用户输入名字。

```
function checkCookie()
{
username=getCookie('username')
if (username!=null && username!="")
  {alert('Welcome again '+username+'!')}
else
  {
  username=prompt('Please enter your name:',"")
  if (username!=null && username!="")
    {
    setCookie('username',username,365)
    }
  }
}
```

这是所有的代码：

```
<html>
<head>
<script type="text/javascript">
function getCookie(c_name)
{
if (document.cookie.length>0)
  {
  c_start=document.cookie.indexOf(c_name + "=")
  if (c_start!=-1)
    {
    c_start=c_start + c_name.length+1
    c_end=document.cookie.indexOf(";",c_start)
    if (c_end==-1) c_end=document.cookie.length
    return unescape(document.cookie.substring(c_start,c_end))
    }
  }
return ""
}

function setCookie(c_name,value,expiredays)
{
var exdate=new Date()
exdate.setDate(exdate.getDate()+expiredays)
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : ";expires="+exdate.toGMTString())
}

function checkCookie()
{
username=getCookie('username')
if (username!=null && username!="")
  {alert('Welcome again '+username+'!')}
else
  {
  username=prompt('Please enter your name:',"")
  if (username!=null && username!="")
    {
    setCookie('username',username,365)
    }
  }
}
</script>
</head>

<body onLoad="checkCookie()">
</body>
</html>
```

HTML DOM

HTML DOM 简介

HTML DOM 定义了访问和操作 **HTML** 文档的标准。

您应该具备的基础知识

在您继续学习之前，您需要对以下内容拥有基本的了解：

- HTML
- CSS
- JavaScript

如果您需要首先学习这些项目，请访问我们的首页。

什么是 DOM？

DOM 是 W3C（万维网联盟）的标准。

DOM 定义了访问 HTML 和 XML 文档的标准：

> “W3C 文档对象模型 （DOM） 是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。”

W3C DOM 标准被分为 3 个不同的部分：

- 核心 DOM - 针对任何结构化文档的标准模型
- XML DOM - 针对 XML 文档的标准模型
- HTML DOM - 针对 HTML 文档的标准模型

编者注：DOM 是 Document Object Model（文档对象模型）的缩写。

什么是 XML DOM？

XML DOM 定义了所有 XML 元素的对象和属性，以及访问它们的方法。

如果您需要学习 XML DOM，请访问我们的 XML DOM 教程。

什么是 HTML DOM？

HTML DOM 是：

- HTML 的标准对象模型
- HTML 的标准编程接口
- W3C 标准

HTML DOM 定义了所有 HTML 元素的对象和属性，以及访问它们的方法。

换言之，*HTML DOM* 是关于如何获取、修改、添加或删除 *HTML* 元素的标准。

HTML DOM 节点

在 **HTML DOM** 中，所有事物都是节点。**DOM** 是被视为节点树的 **HTML**。

DOM 节点

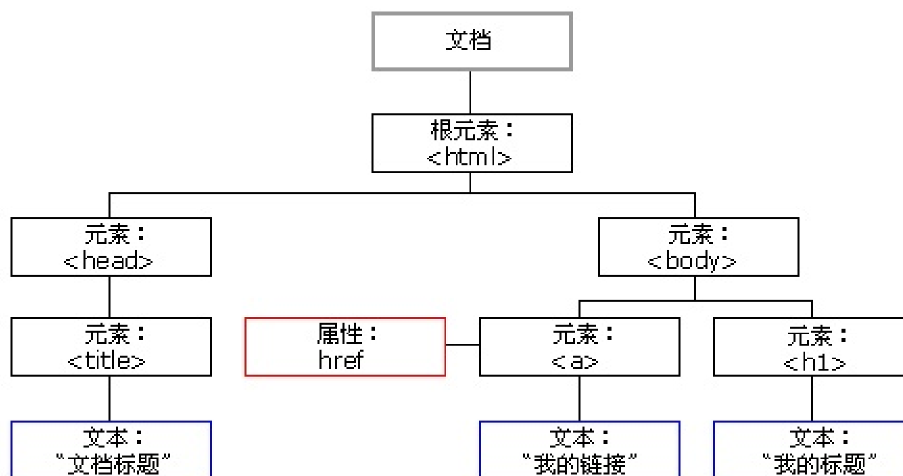
根据 W3C 的 HTML DOM 标准，HTML 文档中的所有内容都是节点：

- 整个文档是一个文档节点
- 每个 HTML 元素是元素节点
- HTML 元素内的文本是文本节点
- 每个 HTML 属性是属性节点
- 注释是注释节点

HTML DOM 节点树

HTML DOM 将 HTML 文档视作树结构。这种结构被称为节点树：

HTML DOM Tree 实例



通过 HTML DOM，树中的所有节点均可通过 JavaScript 进行访问。所有 HTML 元素（节点）均可被修改，也可以创建或删除节点。

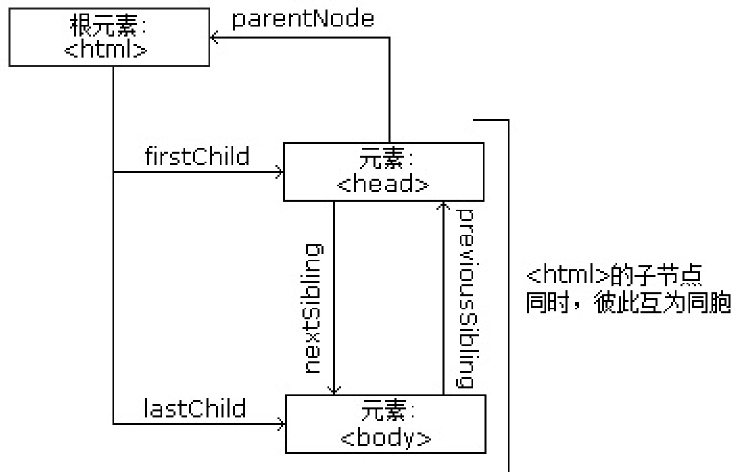
节点父、子和同胞

节点树中的节点彼此拥有层级关系。

父（parent）、子（child）和同胞（sibling）等术语用于描述这些关系。父节点拥有子节点。同级的子节点被称为同胞（兄弟或姐妹）。

- 在节点树中，顶端节点被称为根（root）
- 每个节点都有父节点、除了根（它没有父节点）
- 一个节点可拥有任意数量的子
- 同胞是拥有相同父节点的节点

下面的图片展示了节点树的一部分，以及节点之间的关系：



请看下面的 HTML 片段：

```
<html>
  <head>
    <title>DOM 教程</title>
  </head>
  <body>
    <h1>DOM 第一课</h1>
    <p>Hello world!</p>
  </body>
</html>
```

从上面的 HTML 中：

- `<html>` 节点没有父节点；它是根节点
- `<head>` 和 `<body>` 的父节点是 `<html>` 节点
- 文本节点 "Hello world!" 的父节点是 `<p>` 节点

并且：

- `<html>` 节点拥有两个子节点：`<head>` 和 `<body>`
- `<head>` 节点拥有一个子节点：`<title>` 节点
- `<title>` 节点也拥有一个子节点：文本节点 "DOM 教程"
- `<h1>` 和 `<p>` 节点是同胞节点，同时也是 `<body>` 的子节点

并且：

- `<head>` 元素是 `<html>` 元素的首个子节点
- `<body>` 元素是 `<html>` 元素的最后一个子节点
- `<h1>` 元素是 `<body>` 元素的首个子节点
- `<p>` 元素是 `<body>` 元素的最后一个子节点

警告！

DOM 处理中的常见错误是希望元素节点包含文本。

在本例中：*<title>DOM 教程</title>*，元素节点 <title>，包含值为 "DOM 教程" 的文本节点。

可通过节点的 *innerHTML* 属性来访问文本节点的值。

您将在稍后的章节中学习更多有关 innerHTML 属性的知识。

HTML DOM 方法

方法是我们可以在节点（HTML 元素）上执行的动作。

编程接口

可通过 JavaScript （以及其他编程语言）对 HTML DOM 进行访问。

所有 HTML 元素被定义为对象，而编程接口则是对象方法和对象属性。

方法是您能够执行的动作（比如添加或修改元素）。

属性是您能够获取或设置的值（比如节点的名称或内容）。

getElementById() 方法

getElementById() 方法返回带有指定 ID 的元素：

例子

```
var element=document.getElementById("intro");
```

HTML DOM 对象 - 方法和属性

一些常用的 HTML DOM 方法：

- getElementById(id) - 获取带有指定 id 的节点（元素）
- appendChild(node) - 插入新的子节点（元素）
- removeChild(node) - 删除子节点（元素）

一些常用的 HTML DOM 属性：

- innerHTML - 节点（元素）的文本值
- parentNode - 节点（元素）的父节点
- childNodes - 节点（元素）的子节点
- attributes - 节点（元素）的属性节点

您将在本教程的下一章中学到更多有关属性的知识。

现实生活中的对象

某个人是一个对象。

人的方法可能是 eat(), sleep(), work(), play() 等等。

所有人都有这些方法，但会在不同时间执行它们。

一个人的属性包括姓名、身高、体重、年龄、性别等等。

所有人都有这些属性，但它们的值因人而异。

一些 DOM 对象方法

这里提供一些您将在本教程中学到的常用方法：

方法	描述
getElementById()	返回带有指定 ID 的元素。
getElementsByTagName()	返回包含带有指定标签名称的所有元素的节点列表（集合/节点数组）。
getElementsByClassName()	返回包含带有指定类名的所有元素的节点列表。
appendChild()	把新的子节点添加到指定节点。
removeChild()	删除子节点。
replaceChild()	替换子节点。
insertBefore()	在指定的子节点前面插入新的子节点。
createAttribute()	创建属性节点。
createElement()	创建元素节点。
createTextNode()	创建文本节点。
getAttribute()	返回指定的属性值。
setAttribute()	把指定属性设置或修改为指定的值。

HTML DOM 属性

属性是节点（HTML 元素）的值，您能够获取或设置。

编程接口

可通过 JavaScript （以及其他编程语言）对 HTML DOM 进行访问。

所有 HTML 元素被定义为对象，而编程接口则是对象方法和对象属性。

方法是您能够执行的动作（比如添加或修改元素）。

属性是您能够获取或设置的值（比如节点的名称或内容）。

innerHTML 属性

获取元素内容的最简单方法是使用 innerHTML 属性。

innerHTML 属性对于获取或替换 HTML 元素的内容很有用。

实例

下面的代码获取 id="intro" 的 <p> 元素的 innerHTML：

实例

```
<html>
<body>

<p id="intro">Hello World!</p>

<script>
var txt=document.getElementById("intro").innerHTML;
document.write(txt);
</script>

</body>
</html>
```

在上面的例子中，getElementById 是一个方法，而 innerHTML 是属性。

innerHTML 属性可用于获取或改变任意 HTML 元素，包括 <html> 和 <body>。

nodeName 属性

nodeName 属性规定节点的名称。

- nodeName 是只读的
- 元素节点的 nodeName 与标签名相同
- 属性节点的 nodeName 与属性名相同
- 文本节点的 nodeName 始终是 #text
- 文档节点的 nodeName 始终是 #document

注释：nodeName 始终包含 HTML 元素的大写字母标签名。

nodeValue 属性

nodeValue 属性规定节点的值。

- 元素节点的 nodeValue 是 undefined 或 null
- 文本节点的 nodeValue 是文本本身
- 属性节点的 nodeValue 是属性值

获取元素的值

下面的例子会取回 <p id="intro"> 标签的文本节点值：

实例

```
<html>
<body>

<p id="intro">Hello World!</p>

<script type="text/javascript">
x=document.getElementById("intro");
document.write(x.firstChild.nodeValue);
</script>

</body>
</html>
```

nodeType 属性

nodeType 属性返回节点的类型。nodeType 是只读的。

比较重要的节点类型有：

元素类型	NodeType
元素	1
属性	2
文本	3
注释	8
文档	9

HTML DOM 访问

访问 HTML DOM - 查找 HTML 元素。

访问 HTML 元素（节点）

访问 HTML 元素等同于访问节点

您能够以不同的方式来访问 HTML 元素：

- 通过使用 getElementById() 方法
- 通过使用 getElementsByTagName() 方法
- 通过使用 getElementsByClassName() 方法

getElementById() 方法

getElementById() 方法返回带有指定 ID 的元素：

语法

```
node.getElementById("id");
```

下面的例子获取 id="intro" 的元素：

实例

```
document.getElementById("intro");
```

getElementsByTagName() 方法

getElementsByTagName() 返回带有指定标签名的所有元素。

语法

```
node.getElementsByTagName("tagname");
```

下面的例子返回包含文档中所有 <p> 元素的列表：

实例 1

```
document.getElementsByTagName("p");
```

下面的例子返回包含文档中所有 <p> 元素的列表，并且这些 <p> 元素应该是 id="main" 的元素的后代（子、孙等等）：

实例 2

```
document.getElementById("main").getElementsByTagName("p");
```

getElementsByClassName() 方法

如果您希望查找带有相同类名的所有 HTML 元素，请使用这个方法：

```
document.getElementsByClassName("intro");
```

上面的例子返回包含 class="intro" 的所有元素的一个列表：

注释：getElementsByClassName() 在 Internet Explorer 5,6,7,8 中无效。

HTML DOM - 修改

修改 HTML = 改变元素、属性、样式和事件。

修改 HTML 元素

修改 HTML DOM 意味着许多不同的方面：

- 改变 HTML 内容
- 改变 CSS 样式
- 改变 HTML 属性
- 创建新的 HTML 元素
- 删除已有的 HTML 元素
- 改变事件（处理程序）

在接下来的章节，我们会深入学习修改 HTML DOM 的常用方法。

创建 HTML 内容

改变元素内容的最简答的方法是使用 innerHTML 属性。

下面的例子改变一个 <p> 元素的 HTML 内容：

实例

```
<html>
<body>

<p id="p1">Hello World!</p>

<script>
document.getElementById("p1").innerHTML="New text!";
</script>

</body>
</html>
```

提示：我们将在下面的章节为您解释例子中的细节。

改变 HTML 样式

通过 HTML DOM，您能够访问 HTML 元素的样式对象。

下面的例子改变一个段落的 HTML 样式：

实例

```
<html>

<body>
<p id="p2">Hello world!</p>

<script>
document.getElementById("p2").style.color="blue";
</script>

</body>
</html>
```

创建新的 HTML 元素

如需向 HTML DOM 添加新元素，您首先必须创建该元素（元素节点），然后把它追加到已有的元素上。

实例

```
<div id="d1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para=document.createElement("p");
var node=document.createTextNode("This is new.");
para.appendChild(node);

var element=document.getElementById("d1");
element.appendChild(para);
</script>
```

HTML DOM - 修改 HTML 内容

通过 HTML DOM，JavaScript 能够访问 HTML 文档中的每个元素。

改变 HTML 内容

改变元素内容的最简答的方法是使用 innerHTML 属性。

下面的例子更改 <p> 元素的 HTML 内容：

实例

```
<html>
<body>

<p id="p1">Hello World!</p>

<script>
document.getElementById("p1").innerHTML="New text!";
</script>

</body>
</html>
```

改变 HTML 样式

通过 HTML DOM，您能够访问 HTML 对象的样式对象。

下面的例子更改段落的 HTML 样式：

实例

```
<html>

<body>
<p id="p2">Hello world!</p>

<script>
document.getElementById("p2").style.color="blue";
</script>

</body>
</html>
```

使用事件

HTML DOM 允许您在事件发生时执行代码。

当 HTML 元素”有事情发生“时，浏览器就会生成事件：

- 在元素上点击
- 加载页面
- 改变输入字段

你可以在下一章学习更多有关事件的内容。

下面两个例子在按钮被点击时改变 <body> 元素的背景色：

实例

```
<html>
<body>

<input type="button" onclick="document.body.style.backgroundColor='lavender';"
value="Change background color" />

</body>
</html>
```

在本例中，由函数执行相同的代码：

实例

```
<html>
<body>

<script>
function ChangeBackground()
{
document.body.style.backgroundColor="lavender";
}
</script>

<input type="button" onclick="ChangeBackground()"
value="Change background color" />

</body>
</html>
```

下面的例子在按钮被点击时改变 <p> 元素的文本：

实例

```
<html>
<body>

<p id="p1">Hello world!</p>

<script>
function ChangeText()
{
document.getElementById("p1").innerHTML="New text!";
}
</script>

<input type="button" onclick="ChangeText()" value="Change text">

</body>
</html>
```

HTML DOM - 元素

添加、删除和替换 HTML 元素。

创建新的 HTML 元素 - appendChild()

如需向 HTML DOM 添加新元素，您首先必须创建该元素，然后把它追加到已有的元素上。

实例

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para=document.createElement("p");
var node=document.createTextNode("This is new.");
para.appendChild(node);

var element=document.getElementById("div1");
element.appendChild(para);
</script>
```

例子解释

这段代码创建了一个新的 <p> 元素：

```
var para=document.createElement("p");
```

如需向 <p> 元素添加文本，您首先必须创建文本节点。这段代码创建文本节点：

```
var node=document.createTextNode("This is a new paragraph.");
```

然后您必须向 <p> 元素追加文本节点：

```
para.appendChild(node);
```

最后，您必须向已有元素追加这个新元素。

这段代码查找到一个已有的元素：

```
var element=document.getElementById("div1");
```

这段代码向这个已存在的元素追加新元素：

```
element.appendChild(para);
```

创建新的 HTML 元素 - insertBefore()

上一个例子中的 appendChild() 方法，将新元素作为父元素的最后一个子元素进行添加。

如果不希望如此，您可以使用 insertBefore() 方法：

实例

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para=document.createElement("p");
var node=document.createTextNode("This is new.");
para.appendChild(node);

var element=document.getElementById("div1");
var child=document.getElementById("p1");
element.insertBefore(para,child);
</script>
```

删除已有的 HTML 元素

如需删除 HTML 元素，您必须清楚该元素的父元素：

实例

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
<script>
var parent=document.getElementById("div1");
var child=document.getElementById("p1");
parent.removeChild(child);
</script>
```

例子解释

这个 HTML 文档包含一个带有两个子节点（两个 <p> 元素）的 <div> 元素：

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
```

查找 id="div1" 的元素：

```
var parent=document.getElementById("div1");
```

查找 id="p1" 的 <p> 元素：

```
var child=document.getElementById("p1");
```

从父元素中删除子元素：

```
parent.removeChild(child);
```

提示：能否在不引用父元素的情况下删除某个元素？

很抱歉。DOM 需要了解您需要删除的元素，以及它的父元素。

这里提供一个常用的解决方法：找到您需要删除的子元素，然后使用 parentNode 属性来查找其父元素：

```
var child=document.getElementById("p1");
child.parentNode.removeChild(child);
```

替换 HTML 元素

如需替换 HTML DOM 中的元素，请使用 replaceChild() 方法：

实例

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para=document.createElement("p");
var node=document.createTextNode("This is new.");
para.appendChild(node);

var parent=document.getElementById("div1");
var child=document.getElementById("p1");
parent.replaceChild(para,child);
</script>
```

HTML DOM - 事件

HTML DOM 允许 JavaScript 对 HTML 事件作出反应。。

实例

```
<div style="background-color:#8AB83D;width:160px;height:60px;
margin:20px;padding-top:20px;color:#ffffff;font-weight:bold;
font-size:14px;float:left;text-align:center;"
onmouseover="this.innerHTML='Thank You'"
onmouseout="this.innerHTML='Mouse Over Me'">Mouse Over Me</div>

<div style="background-color:#8AB83D;width:160px;height:60px;
margin:20px;padding-top:20px;color:#ffffff;font-weight:bold;
font-size:14px;float:left;text-align:center;"
onclick="clickMeEvent(this)">Click Me</div>

<script type="application/javascript">
    <!--
    function clickMeEvent(obj)
    {
    if (obj.innerHTML=="Goodbye")
        {
        obj.style.display="none";
        }
    else if (obj.innerHTML=="Thank You")
        {
        obj.innerHTML="Goodbye";
        }
    else if (obj.innerHTML=="Click Me<br>Click Me Again<br>And Again")
        {
        obj.innerHTML="Thank You";
        }
    else if (obj.innerHTML=="Click Me<br>Click Me Again")
        {
        obj.innerHTML=obj.innerHTML+"<br>And Again";
        }
    else
        {
        obj.innerHTML=obj.innerHTML+"<br>Click Me Again";
        }
    }
    //-->
</script>
```

对事件作出反应

当事件发生时，可以执行 JavaScript，比如当用户点击一个 HTML 元素时。

如需在用户点击某个元素时执行代码，请把 JavaScript 代码添加到 HTML 事件属性中：

```
onclick=_JavaScript_
```

HTML 事件的例子：

- 当用户点击鼠标时
- 当网页已加载时
- 当图片已加载时
- 当鼠标移动到元素上时
- 当输入字段被改变时
- 当 HTML 表单被提交时
- 当用户触发按键时

在本例中，当用户点击时，会改变 <h1> 元素的内容：

实例

```
<!DOCTYPE html>
<html>
<body>
<h1 onclick="this.innerHTML='hello!'">请点击这段文本!</h1>
</body>
</html>
```

在本例中，会从事件处理程序中调用函数：

实例

```
<!DOCTYPE html>
<html>
<head>
<script>
function changetext(id)
{
id.innerHTML="hello!";
}
</script>
</head>
<body>
<h1 onclick="changetext(this)">请点击这段文本!</h1>
</body>
</html>
```

HTML 事件属性

如需向 HTML 元素分配事件，您可以使用事件属性。

实例

向 button 元素分配一个 onclick 事件：

```
<button onclick="displayDate()">试一试</button>
```

在上面的例子中，当点击按钮时，会执行名为 displayDate 的函数。

使用 HTML DOM 来分配事件

HTML DOM 允许您使用 JavaScript 向 HTML 元素分配事件：

实例

为 button 元素分配 onclick 事件：

```
<script>
document.getElementById("myBtn").onclick=function(){displayDate()};
</script>
```

在上面的例子中，名为 displayDate 的函数被分配给了 id=myButn" 的 HTML 元素。

当按钮被点击时，将执行函数。

onload 和 onunload 事件

当用户进入或离开页面时，会触发 onload 和 onunload 事件。

onload 事件可用于检查访客的浏览器类型和版本，以便基于这些信息来加载不同版本的网页。

onload 和 onunload 事件可用于处理 cookies。

实例

```
<body onload="checkCookies()">
```

onchange 事件

onchange 事件常用于输入字段的验证。

下面的例子展示了如何使用 onchange。当用户改变输入字段的内容时，将调用 upperCase() 函数。

实例

```
<input type="text" id="fname" onchange="upperCase()">
```

onmouseover 和 onmouseout 事件

onmouseover 和 onmouseout 事件可用于在鼠标指针移动到或离开元素时触发函数。

实例

一个简单的 onmouseover-onmouseout 例子：

```
<div style="background-color:#8AB83D;width:150px;height:60px;
margin:20px;padding-top:20px;color:#ffffff;font-weight:bold;
font-size:18px;text-align:center;" onmouseover="this.innerHTML='谢谢'" onmouseout="this.ir

<p class="tiy"><a target="_blank" href="/tiy/t.asp?f=dom_event_onmouseover">亲自试一试</a><
</div>
```

onmousedown、onmouseup 以及 onclick 事件

onmousedown、onmouseup 以及 onclick 事件是鼠标点击的全部过程。首先当某个鼠标按钮被点击时，触发 onmousedown 事件，然后，当鼠标按钮被松开时，会触发 onmouseup 事件，最后，当鼠标点击完成时，触发 onclick 事件。

实例

一个简单的 onmousedown-onmouseup 实例：

```
<div onmousedown="mDown(this)" onmouseup="mUp(this)" style="background-color:#8AB83D;widt
margin:20px;padding-top:20px;color:#ffffff;font-weight:bold;
font-size:18px;text-align:center;">点击这里</div>

<script type="application/javascript">
    <!--
    function mDown(obj)
    {
    obj.style.backgroundColor="#1ec5e5";
    obj.innerHTML="松开鼠标"
    }

    function mUp(obj)
    {
    obj.style.backgroundColor="#8AB83D";
    obj.innerHTML="谢谢"
    }

    //-->
</script>
```

HTML DOM 事件对象参考手册

如需每个事件的完整描述和例子，请访问我们的 HTML DOM 参考手册。

HTML DOM - 导航

通过 HTML DOM，您能够使用节点关系在节点树中导航。

HTML DOM 节点列表

getElementsByTagName() 方法返回节点列表。节点列表是一个节点数组。

下面的代码选取文档中的所有 <p> 节点：

实例

```
var x=document.getElementsByTagName("p");
```

可以通过下标号访问这些节点。如需访问第二个 <p>，您可以这么写：

```
y=x[1];
```

注释：下标号从 0 开始。

HTML DOM 节点列表长度

length 属性定义节点列表中节点的数量。

您可以使用 length 属性来循环节点列表：

实例

```
x=document.getElementsByTagName("p");

for (i=0;i<x.length;i++)
{
document.write(x[i].innerHTML);
document.write("<br />");
}
```

例子解释：

- 获取所有 `<p>` 元素节点
- 输出每个 `<p>` 元素的文本节点的值

导航节点关系

您能够使用三个节点属性：parentNode、firstChild 以及 lastChild ， 在文档结构中进行导航。

请看下面的 HTML 片段：

```
<html>
<body>

<p>Hello World!</p>
<div>
  <p>DOM 很有用!</p>
  <p>本例演示节点关系。</p>
</div>

</body>
</html>
```

- 首个 `<p>` 元素是 `<body>` 元素的首个子元素（firstChild）
- `<div>` 元素是 `<body>` 元素的最后一个子元素（lastChild）
- `<body>` 元素是首个 `<p>` 元素和 `<div>` 元素的父节点（parentNode）

firstChild 属性可用于访问元素的文本：

实例

```
<html>
<body>

<p id="intro">Hello World!</p>

<script>
x=document.getElementById("intro");
document.write(x.firstChild.nodeValue);
</script>

</body>
</html>
```

DOM 根节点

这里有两个特殊的属性，可以访问全部文档：

- document.documentElement - 全部文档
- document.body - 文档的主体

实例

```
<html>
<body>

<p>Hello World!</p>
<div>
<p>DOM 很有用!</p>
<p>本例演示 <b>document.body</b> 属性。</p>
</div>

<script>
alert(document.body.innerHTML);
</script>

</body>
</html>
```

childNodes 和 nodeValue

除了 innerHTML 属性，您也可以使用 childNodes 和 nodeValue 属性来获取元素的内容。

下面的代码获取 id="intro" 的 <p> 元素的值：

实例

```
<html>
<body>

<p id="intro">Hello World!</p>

<script>
var txt=document.getElementById("intro").childNodes[0].nodeValue;
document.write(txt);
</script>

</body>
</html>
```

在上面的例子中，getElementById 是一个方法，而 childNodes 和 nodeValue 是属性。

在本教程中，我们将使用 innerHTML 属性。不过，学习上面的方法有助于对 DOM 树结构和导航的理解。

JavaScript HTML DOM EventListener

addEventListener() 方法

实例

点用户点击按钮时触发监听事件：

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

addEventListener() 方法用于向指定元素添加事件句柄。

addEventListener() 方法添加的事件句柄不会覆盖已存在的事件句柄。

你可以向一个元素添加多个事件句柄。

你可以向同个元素添加多个同类型的事件句柄，如：两个 "click" 事件。

你可以向任何 DOM 对象添加事件监听，不仅仅是 HTML 元素。如： window 对象。

addEventListener() 方法可以更简单的控制事件（冒泡与捕获）。

当你使用 addEventListener() 方法时, JavaScript 从 HTML 标记中分离开来，可读性更强，在没有控制HTML标记时也可以添加事件监听。

你可以使用 removeEventListener() 方法来移除事件的监听。

语法

element.addEventListener(*event, function, useCapture*);

第一个参数是事件的类型 (如 "click" 或 "mousedown").

第二个参数是事件触发后调用的函数。

第三个参数是个布尔值用于描述事件是冒泡还是捕获。该参数是可选的。

>  注意:不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。

向原元素添加事件句柄

实例

当用户点击元素时弹出 "Hello World!" :

```
_element_.addEventListener("click", function(){ alert("Hello World!"); });
```

你可以使用函数名，来引用外部函数:

实例

当用户点击元素时弹出 "Hello World!" :

```
_element_.addEventListener("click", myFunction);

function myFunction() {
    alert ("Hello World!");
}
```

向同一个元素中添加多个事件句柄

addEventListener() 方法允许向同个元素添加多个事件，且不会覆盖已存在的事件：

实例

```
_element_.addEventListener("click", myFunction);
_element_.addEventListener("click", mySecondFunction);
```

你可以向同个元素添加不同类型的事件：

实例

```
_element_.addEventListener("mouseover", myFunction);
_element_.addEventListener("click", mySecondFunction);
_element_.addEventListener("mouseout", myThirdFunction);
```

向 Window 对象添加事件句柄

addEventListener() 方法允许你在 HTML DOM 对象添加事件监听， HTML DOM 对象如：HTML 元素, HTML 文档, window 对象。或者其他支出的事件对象如: xmlHttpRequest 对象。

实例

当用户重置窗口大小时添加事件监听：

```
window.addEventListener("resize", function(){
    document.getElementById("demo").innerHTML = _sometext_;
});
```

传递参数

当传递参数值时，使用"匿名函数"调用带参数的函数：

实例

```
_element_.addEventListener("click", function(){ myFunction(p1, p2); });
```

事件冒泡或事件捕获？

事件传递有两种方式：冒泡与捕获。

事件传递定义了元素事件触发的顺序。 如果你将 <p> 元素插入到 <div> 元素中，用户点击 <p> 元素，哪个元素的 "click" 事件先被触发呢？

在 冒泡 中，内部元素的事件会先被触发，然后再触发外部元素，即： <p> 元素的点击事件先触发，然后会触发 <div> 元素的点击事件。

在 捕获 中，外部元素的事件会先被触发，然后才会触发内部元素的事件，即： <div> 元素的点击事件先触发 ，然后再触发 <p> 元素的点击事件。

addEventListener() 方法可以指定 "useCapture" 参数来设置传递类型：

addEventListener(*event*, *function*, *useCapture*);

默认值为 false, 及冒泡传递，当值为 true 时, 事件使用捕获传递。

实例

```
document.getElementById("myDiv").addEventListener("click", myFunction, true);
```

removeEventListener() 方法

removeEventListener() 方法移除由 addEventListener() 方法添加的事件句柄:

实例

```
_element_.removeEventListener("mousemove", myFunction);
```

浏览器支持

表格中的数字表示支持该方法的第一个浏览器的版本号。

方法					
addEventListener()	1.0	9.0	1.0	1.0	7.0
removeEventListener()	1.0	9.0	1.0	1.0	7.0

注意： IE 8 及更早 IE 版本，Opera 7.0及其更早版本不支持 addEventListener() 和 removeEventListener() 方法。但是，对于这类浏览器版本可以使用 detachEvent() 方法来移除事件句柄:

element.attachEvent(event, function);
element.detachEvent(event, function);

实例

跨浏览器解决方法:

```
var x = document.getElementById("myBtn");
if (x.addEventListener) {                    // 所有主流浏览器，除了 IE 8 及更早版本
    x.addEventListener("click", myFunction);
} else if (x.attachEvent) {                  // IE 8 及更早版本
    x.attachEvent("onclick", myFunction);
}
```

HTML DOM 事件对象参考手册

所有 HTML DOM 事件，可以查看我们完整的 HTML DOM Event 对象参考手册。

AJAX 教程

AJAX 简介

AJAX 是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。

您应当具备的基础知识

在继续学习之前，您需要对下面的知识有基本的了解：

- HTML / XHTML
- CSS
- JavaScript / DOM

如果您希望首先学习这些项目，请在我们的首页访问这些教程。

什么是 AJAX ？

AJAX = 异步 JavaScript 和 XML。

AJAX 是一种用于创建快速动态网页的技术。

通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

传统的网页（不使用 AJAX）如果需要更新内容，必需重载整个网页面。

有很多使用 AJAX 的应用程序案例：新浪微博、Google 地图、开心网等等。

Google Suggest

在 2005 年，Google 通过其 Google Suggest 使 AJAX 变得流行起来。

Google Suggest 使用 AJAX 创造出动态性极强的 web 界面：当您在谷歌的搜索框输入关键字时，JavaScript 会把这些字符发送到服务器，然后服务器会返回一个搜索建议的列表。

今天就开始使用 AJAX

AJAX 基于已有的标准。这些标准已被大多数开发者使用多年。

请阅读下一章，看看 AJAX 是如何工作的！

AJAX 实例

- XHR 创建对象

为了帮助您理解 **AJAX** 的工作原理，我们创建了一个小型的 **AJAX** 应用程序。

实例

```
<div id="myDiv"><p>AJAX is not a programming language.</p>
<p>It is just a technique for creating better and more interactive web applications.</p><
<button type="button" onclick="loadXMLDoc()">通过 AJAX 改变内容</button>
```

AJAX 实例解释

上面的 AJAX 应用程序包含一个 div 和一个按钮。

div 部分用于显示来自服务器的信息。当按钮被点击时，它负责调用名为 loadXMLDoc() 的函数：

```
<html>
<body>

<div id="myDiv"><h3>Let AJAX change this text</h3></div>
<button type="button" onclick="loadXMLDoc()">Change Content</button>

</body>
</html>
```

接下来，在页面的 head 部分添加一个 <script> 标签。该标签中包含了这个 loadXMLDoc() 函数：

```
<head>
<script type="text/javascript">
function loadXMLDoc()
{
.... AJAX script goes here ...
}
</script>
</head>
```

下面的章节会为您讲解 AJAX 的工作原理。

- XHR 创建对象

AJAX - 创建 XMLHttpRequest 对象

- XHR 请求

XMLHttpRequest 是 AJAX 的基础。

XMLHttpRequest 对象

所有现代浏览器均支持 XMLHttpRequest 对象（IE5 和 IE6 使用 ActiveXObject）。

XMLHttpRequest 用于在后台与服务器交换数据。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

创建 XMLHttpRequest 对象

所有现代浏览器（IE7+、Firefox、Chrome、Safari 以及 Opera）均内建 XMLHttpRequest 对象。

创建 XMLHttpRequest 对象的语法：

```
variable=new XMLHttpRequest();
```

老版本的 Internet Explorer （IE5 和 IE6）使用 ActiveX 对象：

```
variable=new ActiveXObject("Microsoft.XMLHTTP");
```

为了应对所有的现代浏览器，包括 IE5 和 IE6，请检查浏览器是否支持 XMLHttpRequest 对象。如果支持，则创建 XMLHttpRequest 对象。如果不支持，则创建 ActiveXObject ：

```
var xmlhttp;
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
```

在下一章中，您将学习发送服务器请求的知识。

- XHR 请求

AJAX - 向服务器发送请求

- XHR 创建对象
- XHR 响应

XMLHttpRequest 对象用于和服务器交换数据。

向服务器发送请求

如需将请求发送到服务器，我们使用 XMLHttpRequest 对象的 open() 和 send() 方法：

```
xmlhttp.open("GET","test1.txt",true);
xmlhttp.send();
```

方法	描述
open(*method*,*url*,*async*)	规定请求的类型、URL 以及是否异步处理请求。 *method*：请求的类型；GET 或 POST *url*：文件在服务器上的位置 *async*：true（异步）或 false（同步）
send(*string*)	将请求发送到服务器。 *string*：仅用于 POST 请求

GET 还是 POST？

与 POST 相比，GET 更简单也更快，并且在大部分情况下都能用。

然而，在以下情况中，请使用 POST 请求：

- 无法使用缓存文件（更新服务器上的文件或数据库）
- 向服务器发送大量数据（POST 没有数据量限制）
- 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

GET 请求

一个简单的 GET 请求：

```
xmlhttp.open("GET","demo_get.asp",true);
xmlhttp.send();
```

在上面的例子中，您可能得到的是缓存的结果。

为了避免这种情况，请向 URL 添加一个唯一的 ID：

```
xmlhttp.open("GET","demo_get.asp?t=" + `Math.random()`,true);
xmlhttp.send();
```

如果您希望通过 GET 方法发送信息，请向 URL 添加信息：

```
xmlhttp.open("GET","demo_get2.asp?fname=Bill&lname=Gates",true);
xmlhttp.send();
```

POST 请求

一个简单 POST 请求：

```
xmlhttp.open("POST","demo_post.asp",true);
xmlhttp.send();
```

如果需要像 HTML 表单那样 POST 数据，请使用 setRequestHeader() 来添加 HTTP 头。然后在 send() 方法中规定您希望发送的数据：

```
xmlhttp.open("POST","ajax_test.asp",true);
xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
xmlhttp.send("fname=Bill&lname=Gates");
```

方法	描述
setRequestHeader(*header*,*value*)	向请求添加 HTTP 头。 *header*: 规定头的名称 *value*: 规定头的值

url - 服务器上的文件

open() 方法的 *url* 参数是服务器上文件的地址：

```
xmlhttp.open("GET","ajax_test.asp",true);
```

该文件可以是任何类型的文件，比如 .txt 和 .xml，或者服务器脚本文件，比如 .asp 和 .php（在传回响应之前，能够在服务器上执行任务）。

异步 - True 或 False？

AJAX 指的是异步 JavaScript 和 XML（Asynchronous JavaScript and XML）。

XMLHttpRequest 对象如果要用于 AJAX 的话，其 open() 方法的 async 参数必须设置为 true：

```
xmlhttp.open("GET","ajax_test.asp",`true`);
```

对于 web 开发人员来说，发送异步请求是一个巨大的进步。很多在服务器执行的任务都相当费时。AJAX 出现之前，这可能会引起应用程序挂起或停止。

通过 AJAX，JavaScript 无需等待服务器的响应，而是：

- 在等待服务器响应时执行其他脚本
- 当响应就绪后对响应进行处理

Async = true

当使用 async=true 时，请规定在响应处于 onreadystatechange 事件中的就绪状态时执行的函数：

```
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
  }
xmlhttp.open("GET","test1.txt",true);
xmlhttp.send();
```

您将在稍后的章节学习更多有关 onreadystatechange 的内容。

Async = false

如需使用 async=false，请将 open() 方法中的第三个参数改为 false：

```
xmlhttp.open("GET","test1.txt",false);
```

我们不推荐使用 async=false，但是对于一些小型的请求，也是可以的。

请记住，JavaScript 会等到服务器响应就绪才继续执行。如果服务器繁忙或缓慢，应用程序会挂起或停止。

注释：当您使用 async=false 时，请不要编写 onreadystatechange 函数 - 把代码放到 send() 语句后面即可：

```
xmlhttp.open("GET","test1.txt",false);
xmlhttp.send();
document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
```

- XHR 创建对象
- XHR 响应

AJAX - 服务器响应

- XHR 请求
- XHR readyState

服务器响应

如需获得来自服务器的响应，请使用 XMLHttpRequest 对象的 responseText 或 responseXML 属性。

属性	描述
responseText	获得字符串形式的响应数据。
responseXML	获得 XML 形式的响应数据。

responseText 属性

如果来自服务器的响应并非 XML，请使用 responseText 属性。

responseText 属性返回字符串形式的响应，因此您可以这样使用：

```
document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
```

responseXML 属性

如果来自服务器的响应是 XML，而且需要作为 XML 对象进行解析，请使用 responseXML 属性：

请求 books.xml 文件，并解析响应：

```
xmlDoc=xmlhttp.responseXML;
txt="";
x=xmlDoc.getElementsByTagName("ARTIST");
for (i=0;i<x.length;i++)
  {
  txt=txt + x[i].childNodes[0].nodeValue + "<br />";
  }
document.getElementById("myDiv").innerHTML=txt;
```

- XHR 请求
- XHR readyState

AJAX - onreadystatechange 事件

- XHR 响应

onreadystatechange 事件

当请求被发送到服务器时，我们需要执行一些基于响应的任务。

每当 readyState 改变时，就会触发 onreadystatechange 事件。

readyState 属性存有 XMLHttpRequest 的状态信息。

下面是 XMLHttpRequest 对象的三个重要的属性：

属性	描述
onreadystatechange	存储函数（或函数名），每当 readyState 属性改变时，就会调用该函数。
readyState	存有 XMLHttpRequest 的状态。从 0 到 4 发生变化。 0: 请求未初始化 1: 服务器连接已建立 2: 请求已接收 3: 请求处理中 4: 请求已完成，且响应已就绪
status	200: "OK" 404: 未找到页面

在 onreadystatechange 事件中，我们规定当服务器响应已做好被处理的准备时所执行的任务。

当 readyState 等于 4 且状态为 200 时，表示响应已就绪：

```
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
  }
```

注释：onreadystatechange 事件被触发 5 次（0 - 4），对应着 readyState 的每个变化。

使用 Callback 函数

callback 函数是一种以参数形式传递给另一个函数的函数。

如果您的网站上存在多个 AJAX 任务，那么您应该为创建 XMLHttpRequest 对象编写一个标准的函数，并为每个 AJAX 任务调用该函数。

该函数调用应该包含 URL 以及发生 onreadystatechange 事件时执行的任务（每次调用可能不尽相同）：

```
function myFunction()
{
loadXMLDoc("ajax_info.txt",function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
  });
}
```

- XHR 响应

AJAX ASP/PHP 请求实例

- XHR readyState

AJAX 用于创造动态性更强的应用程序。

AJAX ASP/PHP 实例

下面的例子将为您演示当用户在输入框中键入字符时，网页如何与 web 服务器进行通信：

```
<p>请在下面的输入框中键入字母（A - Z）：</p>

<form action="" style="margin-top:15px;">
姓氏：<input id="txt1" onkeyup="showHint(this.value)" type="text">
</form>

<p>建议：<span id="txtHint"></span></p>
```

实例解释 - showHint() 函数

当用户在上面的输入框中键入字符时，会执行函数 "showHint()" 。该函数由 "onkeyup" 事件触发：

```
function showHint(str)
{
var xmlhttp;
if (str.length==0)
  {
  document.getElementById("txtHint").innerHTML="";
  return;
  }
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("txtHint").innerHTML=xmlhttp.responseText;
    }
  }
xmlhttp.open("GET","gethint.asp?q="+str,true);
xmlhttp.send();
}
```

源代码解释：

如果输入框为空 (str.length==0)，则该函数清空 txtHint 占位符的内容，并退出函数。

如果输入框不为空，showHint() 函数执行以下任务：

- 创建 XMLHttpRequest 对象
- 当服务器响应就绪时执行函数
- 把请求发送到服务器上的文件
- 请注意我们向 URL 添加了一个参数 q （带有输入框的内容）

AJAX 服务器页面 - ASP 和 PHP

由上面的 JavaScript 调用的服务器页面是 ASP 文件，名为 "gethint.asp"。

下面，我们创建了两个版本的服务器文件，一个用 ASP 编写，另一个用 PHP 编写。

ASP 文件

"gethint.asp" 中的源代码会检查一个名字数组，然后向浏览器返回相应的名字：

```
<%
response.expires=-1
dim a(30)
'用名字来填充数组
a(1)="Anna"
a(2)="Brittany"
a(3)="Cinderella"
a(4)="Diana"
a(5)="Eva"
a(6)="Fiona"
a(7)="Gunda"
a(8)="Hege"
a(9)="Inga"
a(10)="Johanna"
a(11)="Kitty"
a(12)="Linda"
a(13)="Nina"
a(14)="Ophelia"
a(15)="Petunia"
a(16)="Amanda"
a(17)="Raquel"
a(18)="Cindy"
a(19)="Doris"
a(20)="Eve"
a(21)="Evita"
a(22)="Sunniva"
a(23)="Tove"
a(24)="Unni"
a(25)="Violet"
a(26)="Liza"
a(27)="Elizabeth"
a(28)="Ellen"
a(29)="Wenche"
a(30)="Vicky"

'获得来自 URL 的 q 参数
q=ucase(request.querystring("q"))

'如果 q 大于 0，则查找数组中的所有提示
if len(q)>0 then
  hint=""
  for i=1 to 30
    if q=ucase(mid(a(i),1,len(q))) then
      if hint="" then
        hint=a(i)
      else
        hint=hint & " , " & a(i)
      end if
    end if
  next
end if

'如果未找到提示，则输出 "no suggestion"
'否则输出正确的值
if hint="" then
  response.write("no suggestion")
else
  response.write(hint)
end if
%>
```

PHP 文件

下面的代码用 PHP 编写，与上面的 ASP 代码作用是一样的。

注释：如需在 PHP 中运行这个例子，请将 url 变量的值（Javascript 代码中）由 "gethint.asp" 改为 "gethint.php"。

```
<?php
// 用名字来填充数组
$a[]="Anna";
$a[]="Brittany";
$a[]="Cinderella";
$a[]="Diana";
$a[]="Eva";
$a[]="Fiona";
$a[]="Gunda";
$a[]="Hege";
$a[]="Inga";
$a[]="Johanna";
$a[]="Kitty";
$a[]="Linda";
$a[]="Nina";
$a[]="Ophelia";
$a[]="Petunia";
$a[]="Amanda";
$a[]="Raquel";
$a[]="Cindy";
$a[]="Doris";
$a[]="Eve";
$a[]="Evita";
$a[]="Sunniva";
$a[]="Tove";
$a[]="Unni";
$a[]="Violet";
$a[]="Liza";
$a[]="Elizabeth";
$a[]="Ellen";
$a[]="Wenche";
$a[]="Vicky";

//获得来自 URL 的 q 参数
$q=$_GET["q"];

//如果 q 大于 0，则查找数组中的所有提示
if (strlen($q) > 0)
  {
  $hint="";
  for($i=0; $i<count($a); $i++)
    {
    if (strtolower($q)==strtolower(substr($a[$i],0,strlen($q))))
      {
      if ($hint=="")
        {
        $hint=$a[$i];
        }
      else
        {
        $hint=$hint." , ".$a[$i];
        }
      }
    }
  }

// 如果未找到提示，则把输出设置为 "no suggestion"
// 否则设置为正确的值
if ($hint == "")
  {
  $response="no suggestion";
  }
else
  {
  $response=$hint;
  }

//输出响应
echo $response;
?>
```

- XHR readyState

AJAX 数据库实例

AJAX 可用来与数据库进行动态通信。

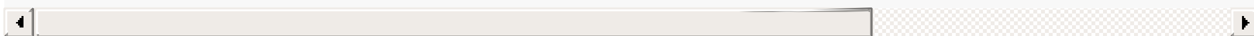
AJAX 数据库实例

下面的例子将演示网页如何通过 AJAX 从数据库读取信息：

```
<p>请在下面的下拉列表中选择一个客户：</p>

<form action="" style="margin-top:15px;">
<label>请选择一位客户：
<select name="customers" onchange="showCustomer(this.value)" style="font-family:Verdana,
<option value="APPLE">Apple Computer, Inc.</option>
<option value="BAIDU ">BAIDU, Inc</option>
<option value="Canon">Canon USA, Inc.</option>
<option value="Google">Google, Inc.</option>
<option value="Nokia">Nokia Corporation</option>
<option value="SONY">Sony Corporation of America</option>
</select>
</label>
</form>

<div id="txtHint" style="margin:15px 0 0 0;padding:0;border:0;color:#0479A7; font-weight:
```



实例解释 - showCustomer() 函数

当用户在上面的下拉列表中选择某个客户时，会执行名为 "showCustomer()" 的函数。该函数由 "onchange" 事件触发：

```
function showCustomer(str)
{
var xmlhttp;
if (str=="")
  {
  document.getElementById("txtHint").innerHTML="";
  return;
  }
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("txtHint").innerHTML=xmlhttp.responseText;
    }
  }
xmlhttp.open("GET","getcustomer.asp?q="+str,true);
xmlhttp.send();
}
```

showCustomer() 函数执行以下任务：

- 检查是否已选择某个客户
- 创建 XMLHttpRequest 对象
- 当服务器响应就绪时执行所创建的函数
- 把请求发送到服务器上的文件
- 请注意我们向 URL 添加了一个参数 q （带有输入域中的内容）

AJAX 服务器页面

由上面的 JavaScript 调用的服务器页面是 ASP 文件，名为 "getcustomer.asp"。

用 PHP 编写服务器文件也很容易，或者用其他服务器语言。请看用 PHP 编写的相应的例子。

"getcustomer.asp" 中的源代码负责对数据库进行查询，然后用 HTML 表格返回结果：

```
<%
response.expires=-1
sql="SELECT * FROM CUSTOMERS WHERE CUSTOMERID="
sql=sql & "'" & request.querystring("q") & "'"

set conn=Server.CreateObject("ADODB.Connection")
conn.Provider="Microsoft.Jet.OLEDB.4.0"
conn.Open(Server.Mappath("/db/northwind.mdb"))
set rs=Server.CreateObject("ADODB.recordset")
rs.Open sql,conn

response.write("<table>")
do until rs.EOF
  for each x in rs.Fields
    response.write("<tr><td><b>" & x.name & "</b></td>")
    response.write("<td>" & x.value & "</td></tr>")
  next
  rs.MoveNext
loop
response.write("</table>")
%>
```

AJAX XML 实例

AJAX 可用来与 **XML** 文件进行交互式通信。

AJAX XML 实例

下面的例子将演示网页如何使用 AJAX 来读取来自 XML 文件的信息：

```
<div id="txtCDInfo" style="margin:15px 0 0 0; border:0; padding:0;">
<button onclick="loadXMLDoc('/example/xmle/cd_catalog.xml')" style="font-family:Verdana,
</div>
```



实例解释 - loadXMLDoc() 函数

当用户点击上面的"获得 CD 信息"这个按钮，就会执行 loadXMLDoc() 函数。

loadXMLDoc() 函数创建 XMLHttpRequest 对象，添加当服务器响应就绪时执行的函数，并将请求发送到服务器。

当服务器响应就绪时，会构建一个 HTML 表格，从 XML 文件中提取节点（元素），最后使用已经填充了 XML 数据的 HTML 表格来更新 txtCDInfo 占位符：

```
function loadXMLDoc(url)
{
var xmlhttp;
var txt,xx,x,i;
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    txt="<table border='1'><tr><th>Title</th><th>Artist</th></tr>";
    x=xmlhttp.responseXML.documentElement.getElementsByTagName("CD");
    for (i=0;i<x.length;i++)
      {
      txt=txt + "<tr>";
      xx=x[i].getElementsByTagName("TITLE");
        {
        try
          {
          txt=txt + "<td>" + xx[0].firstChild.nodeValue + "</td>";
          }
        catch (er)
          {
          txt=txt + "<td> </td>";
          }
        }
    xx=x[i].getElementsByTagName("ARTIST");
      {
        try
          {
          txt=txt + "<td>" + xx[0].firstChild.nodeValue + "</td>";
          }
        catch (er)
          {
          txt=txt + "<td> </td>";
          }
        }
      txt=txt + "</tr>";
      }
    txt=txt + "</table>";
    document.getElementById('txtCDInfo').innerHTML=txt;
    }
  }
xmlhttp.open("GET",url,true);
xmlhttp.send();
}
```

AJAX 服务器页面

上面这个例子中使用的服务器页面实际上是一个 XML 文件，名为 "cd_catalog.xml"。

jQuery 基础

jQuery 简介

jQuery 库可以通过一行简单的标记被添加到网页中。

jQuery 库 - 特性

jQuery 是一个 JavaScript 函数库。

jQuery 库包含以下特性：

- HTML 元素选取
- HTML 元素操作
- CSS 操作
- HTML 事件函数
- JavaScript 特效和动画
- HTML DOM 遍历和修改
- AJAX
- Utilities

向您的页面添加 jQuery 库

jQuery 库位于一个 JavaScript 文件中，其中包含了所有的 jQuery 函数。

可以通过下面的标记把 jQuery 添加到网页中：

```
<head>
`<script type="text/javascript" src="jquery.js"></script>`
</head>
```

请注意，<script> 标签应该位于页面的 <head> 部分。

基础 jQuery 实例

下面的例子演示了 jQuery 的 hide() 函数，隐藏了 HTML 文档中所有的 <p> 元素。

实例

```
<html>
<head>
`<script type="text/javascript" src="jquery.js"></script>`
<script type="text/javascript">
$(document).ready(function(){
$("button").click(function(){
`$("p").hide();`
});
});
</script>
</head>

<body>
<h2>This is a heading</h2>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
<button type="button">Click me</button>
</body>
</html>
```

下载 jQuery

共有两个版本的 jQuery 可供下载：一份是精简过的，另一份是未压缩的（供调试或阅读）。

这两个版本都可从 jQuery.com 下载。

库的替代

Google 和 Microsoft 对 jQuery 的支持都很好。

如果您不愿意在自己的计算机上存放 jQuery 库，那么可以从 Google 或 Microsoft 加载 CDN jQuery 核心文件。

使用 Google 的 CDN

```
<head>
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs
/jquery/1.4.0/jquery.min.js"></script>
</head>
```

使用 Microsoft 的 CDN

```
<head>
<script type="text/javascript" src="http://ajax.microsoft.com/ajax/jquery
/jquery-1.4.min.js"></script>
</head>
```

jQuery 安装

把 jQuery 添加到您的网页

如需使用 jQuery，您需要下载 jQuery 库（会在下面为您讲解），然后把它包含在希望使用的网页中。

jQuery 库是一个 JavaScript 文件，您可以使用 HTML 的 <script> 标签引用它：

```
<head>
<script src="jquery.js"></script>
</head>
```

请注意，<script> 标签应该位于页面的 <head> 部分。

提示：您是否很疑惑为什么我们没有在 <script> 标签中使用 type="text/javascript" ？

在 HTML5 中，不必那样做了。JavaScript 是 HTML5 以及所有现代浏览器中的默认脚本语言！

下载 jQuery

有两个版本的 jQuery 可供下载：

- Production version - 用于实际的网站中，已被精简和压缩。
- Development version - 用于测试和开发（未压缩，是可读的代码）

这两个版本都可以从 jQuery.com 下载。

提示：您可以把下载文件放到与页面相同的目录中，这样更方便使用。

替代方案

如果您不希望下载并存放 jQuery，那么也可以通过 CDN（内容分发网络） 引用它。

谷歌和微软的服务器都存有 jQuery 。

如需从谷歌或微软引用 jQuery，请使用以下代码之一：

Google CDN:

```
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js">
</script>
</head>
```

提示：通过 Google CDN 来获得最新可用的版本：

如果您观察上什么的 Google URL - 在 URL 中规定了 jQuery 版本 (1.8.0)。如果您希望使用最新版本的 jQuery，也可以从版本字符串的末尾（比如本例 1.8）删除一个数字，谷歌会返回 1.8 系列中最新的可用版本（1.8.0、1.8.1 等等），或者也可以只剩第一个数字，那么谷歌会返回 1 系列中最新的可用版本（从 1.1.0 到 1.9.9）。

Microsoft CDN:

```
<head>
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.0.js">
</script>
</head>
```

提示：使用谷歌或微软的 jQuery，有一个很大的优势：

许多用户在访问其他站点时，已经从谷歌或微软加载过 jQuery。所有结果是，当他们访问您的站点时，会从缓存中加载 jQuery，这样可以减少加载时间。同时，大多数 CDN 都可以确保当用户向其请求文件时，会从离用户最近的服务器上返回响应，这样也可以提高加载速度。

jQuery 语法

通过 **jQuery**，您可以选取（查询，**query**） **HTML** 元素，并对它们执行“操作”（**actions**）。

jQuery 语法实例

$(this).hide()

演示 jQuery hide() 函数，隐藏当前的 HTML 元素。

```
<html>
<head>
<script type="text/javascript" src="/jquery/jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $("button").click(function(){
  $(this).hide();
});
});
</script>
</head>

<body>
<button type="button">Click me</button>
</body>

</html>
```

$("#test").hide()

演示 jQuery hide() 函数，隐藏 id="test" 的元素。

```
<html>
<head>
<script type="text/javascript" src="/jquery/jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $("button").click(function(){
    $("#test").hide();
  });
});
</script>
</head>

<body>
<h2>This is a heading</h2>
<p>This is a paragraph.</p>
<p id="test">This is another paragraph.</p>
<button type="button">Click me</button>
</body>

</html>
```

$("p").hide()

演示 jQuery hide() 函数，隐藏所有

元素。

```
<html>
<head>
<script type="text/javascript" src="/jquery/jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
$("button").click(function(){
$("p").hide();
});
});
</script>
</head>

<body>
<h2>This is a heading</h2>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
<button type="button">Click me</button>
</body>
</html>
```

$(".test").hide()

演示 jQuery hide() 函数，隐藏所有 class="test" 的元素。

```
<html>
<head>
<script type="text/javascript" src="/jquery/jquery.js"></script>
<script type="text/javascript">
$(document).ready(function()
{
  $("button").click(function()
  {
  $(".test").hide();
  });
});
</script>
</head>
<body>

<h2 class="test">This is a heading</h2>
<p class="test">This is a paragraph.</p>
<p>This is another paragraph.</p>
<button type="button">Click me</button>

</body>
</html>
```

jQuery 语法

jQuery 语法是为 HTML 元素的选取编制的，可以对元素执行某些操作。

基础语法是：*$(selector).action()*

- 美元符号定义 jQuery
- 选择符（selector）“查询”和“查找” HTML 元素
- jQuery 的 action() 执行对元素的操作

示例

$(this).hide() - 隐藏当前元素

$("p").hide() - 隐藏所有段落

$(".test").hide() - 隐藏所有 class="test" 的所有元素

$("#test").hide() - 隐藏所有 id="test" 的元素

提示：jQuery 使用的语法是 XPath 与 CSS 选择器语法的组合。在本教程接下来的章节，您将学习到更多有关选择器的语法。

文档就绪函数

您也许已经注意到在我们的实例中的所有 jQuery 函数位于一个 document ready 函数中：

```
$(document).ready(function(){

--- jQuery functions go here ----

});
```

这是为了防止文档在完全加载（就绪）之前运行 jQuery 代码。

如果在文档没有完全加载之前就运行函数，操作可能失败。下面是两个具体的例子：

- 试图隐藏一个不存在的元素
- 获得未完全加载的图像的大小

jQuery 选择器

选择器允许您对元素组或单个元素进行操作。

jQuery 选择器

在前面的章节中，我们展示了一些有关如何选取 HTML 元素的实例。

关键点是学习 jQuery 选择器是如何准确地选取您希望应用效果的元素。

jQuery 元素选择器和属性选择器允许您通过标签名、属性名或内容对 HTML 元素进行选择。

选择器允许您对 HTML 元素组或单个元素进行操作。

在 HTML DOM 术语中：

选择器允许您对 DOM 元素组或单个 DOM 节点进行操作。

jQuery 元素选择器

jQuery 使用 CSS 选择器来选取 HTML 元素。

$("p") 选取 <p> 元素。

$("p.intro") 选取所有 class="intro" 的 <p> 元素。

$("p#demo") 选取所有 id="demo" 的 <p> 元素。

jQuery 属性选择器

jQuery 使用 XPath 表达式来选择带有给定属性的元素。

$("[href]") 选取所有带有 href 属性的元素。

$("[href='#']") 选取所有带有 href 值等于 "#" 的元素。

$("[href!='#']") 选取所有带有 href 值不等于 "#" 的元素。

$("[href$='.jpg']") 选取所有 href 值以 ".jpg" 结尾的元素。

jQuery CSS 选择器

jQuery CSS 选择器可用于改变 HTML 元素的 CSS 属性。

下面的例子把所有 p 元素的背景颜色更改为红色：

实例

```
$("p").css("background-color","red");
```

更多的选择器实例

语法	描述
$(this)	当前 HTML 元素
$("p")	所有 `<p>` 元素
$("p.intro")	所有 class="intro" 的 `<p>` 元素
$(".intro")	所有 class="intro" 的元素
$("#intro")	id="intro" 的元素
$("ul li:first")	每个 `<ul>` 的第一个 `<li>` 元素
$("[href$='.jpg']")	所有带有以 ".jpg" 结尾的属性值的 href 属性
$("div#intro .head")	id="intro" 的 `<div>` 元素中的所有 class="head" 的元素

如需完整的参考手册，请访问我们的 jQuery 选择器参考手册。

jQuery 事件

jQuery 是为事件处理特别设计的。

jQuery 事件函数

jQuery 事件处理方法是 jQuery 中的核心函数。

事件处理程序指的是当 HTML 中发生某些事件时所调用的方法。术语由事件“触发”（或“激发”）经常会被使用。

通常会把 jQuery 代码放到 <head>部分的事件处理方法中：

实例

```
<html>
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  `$("button").click(function(){
    $("p").hide();
  });`
});
</script>
</head>

<body>
<h2>This is a heading</h2>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
<button>Click me</button>
</body>

</html>
```

在上面的例子中，当按钮的点击事件被触发时会调用一个函数：

```
$("button").click(function() {..some code... } )
```

该方法隐藏所有 <p> 元素：

```
$("p").hide();
```

单独文件中的函数

如果您的网站包含许多页面，并且您希望您的 jQuery 函数易于维护，那么请把您的 jQuery 函数放到独立的 .js 文件中。

当我们在教程中演示 jQuery 时，会将函数直接添加到 <head> 部分中。不过，把它们放到一个单独的文件中会更好，就像这样（通过 src 属性来引用文件）：

实例

```
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="my_jquery_functions.js"></script>
</head>
```

jQuery 名称冲突

jQuery 使用 $ 符号作为 jQuery 的简介方式。

某些其他 JavaScript 库中的函数（比如 Prototype）同样使用 $ 符号。

jQuery 使用名为 noConflict() 的方法来解决该问题。

var jq=jQuery.noConflict()，帮助您使用自己的名称（比如 jq）来代替 $ 符号。

结论

由于 jQuery 是为处理 HTML 事件而特别设计的，那么当您遵循以下原则时，您的代码会更恰当且更易维护：

- 把所有 jQuery 代码置于事件处理函数中
- 把所有事件处理函数置于文档就绪事件处理器中
- 把 jQuery 代码置于单独的 .js 文件中
- 如果存在名称冲突，则重命名 jQuery 库

jQuery 事件

下面是 jQuery 中事件方法的一些例子：

Event 函数	绑定函数至
$(document).ready(function)	将函数绑定到文档的就绪事件（当文档完成加载时）
$(selector).click(function)	触发或将函数绑定到被选元素的点击事件
$(selector).dblclick(function)	触发或将函数绑定到被选元素的双击事件
$(selector).focus(function)	触发或将函数绑定到被选元素的获得焦点事件
$(selector).mouseover(function)	触发或将函数绑定到被选元素的鼠标悬停事件

如需完整的参考手册，请访问我们的 jQuery 事件参考手册。

jQuery 效果

jQuery 效果 - 隐藏和显示

隐藏、显示、切换，滑动，淡入淡出，以及动画，哇哦！

实例

jQuery hide()

演示一个简单的 jQuery hide() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js">
</script>
<script>
$(document).ready(function(){
  $("p").click(function(){
    $(this).hide();
  });
});
</script>
</head>
<body>
<p>如果您点击我，我会消失。</p>
<p>点击我，我会消失。</p>
<p>也要点击我哦。</p>
</body>
</html>
```

jQuery hide()

另一个 hide() 演示。如何隐藏部分文本。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $(".ex .hide").click(function(){
    $(this).parents(".ex").hide("slow");
  });
});
</script>
<style type="text/css">
div.ex
{
background-color:#e5eecc;
padding:7px;
border:solid 1px #c3c3c3;
}
</style>
</head>

<body>

<h3>中国办事处</h3>
<div class="ex">
<button class="hide" type="button">隐藏</button>
<p>联系人：张先生<br />
北三环中路 100 号<br />
北京</p>
</div>

<h3>美国办事处</h3>
<div class="ex">
<button class="hide" type="button">隐藏</button>
<p>联系人：David<br />
第五大街 200 号<br />
纽约</p>
</div>

</body>
</html>
```

jQuery hide() 和 show()

通过 jQuery，您可以使用 hide() 和 show() 方法来隐藏和显示 HTML 元素：

```
$("#hide").click(function(){
  $("p").hide();
});

$("#show").click(function(){
  $("p").show();
});
```

语法：

```
$(selector).hide(speed,callback);

$(selector).show(speed,callback);
```

可选的 speed 参数规定隐藏/显示的速度，可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是隐藏或显示完成后所执行的函数名称。

下面的例子演示了带有 speed 参数的 hide() 方法：

实例

```
$("button").click(function(){
  $("p").hide(1000);
});
```

jQuery toggle()

通过 jQuery，您可以使用 toggle() 方法来切换 hide() 和 show() 方法。

显示被隐藏的元素，并隐藏已显示的元素：

实例

```
$("button").click(function(){
  $("p").toggle();
});
```

语法：

```
$(selector).toggle(speed,callback);
```

可选的 speed 参数规定隐藏/显示的速度，可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是 toggle() 方法完成后所执行的函数名称。

jQuery 效果参考手册

如需全面查阅 jQuery 效果，请访问我们的 jQuery 效果参考手册。

jQuery 效果 - 淡入淡出

通过 **jQuery**，您可以实现元素的淡入淡出效果。

实例

jQuery fadeIn()

演示 jQuery fadeIn() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").fadeIn();
    $("#div2").fadeIn("slow");
    $("#div3").fadeIn(3000);
  });
});
</script>
</head>

<body>
<p>演示带有不同参数的 fadeIn() 方法。</p>
<button>点击这里，使三个矩形淡入</button>
<br><br>
<div id="div1" style="width:80px;height:80px;display:none;background-color:red;"></div>
<br>
<div id="div2" style="width:80px;height:80px;display:none;background-color:green;"></div>
<br>
<div id="div3" style="width:80px;height:80px;display:none;background-color:blue;"></div>
</body>
</html>
```

jQuery fadeOut()

演示 jQuery fadeOut() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").fadeOut();
    $("#div2").fadeOut("slow");
    $("#div3").fadeOut(3000);
  });
});
</script>
</head>

<body>
<p>演示带有不同参数的 fadeOut() 方法。</p>
<button>点击这里，使三个矩形淡出</button>
<br><br>
<div id="div1" style="width:80px;height:80px;background-color:red;"></div>
<br>
<div id="div2" style="width:80px;height:80px;background-color:green;"></div>
<br>
<div id="div3" style="width:80px;height:80px;background-color:blue;"></div>
</body>

</html>
```

jQuery fadeToggle()

演示 jQuery fadeToggle() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").fadeToggle();
    $("#div2").fadeToggle("slow");
    $("#div3").fadeToggle(3000);
  });
});
</script>
</head>

<body>

<p>演示带有不同参数的 fadeToggle() 方法。</p>
<button>点击这里，使三个矩形淡入淡出</button>
<br><br>
<div id="div1" style="width:80px;height:80px;background-color:red;"></div>
<br>
<div id="div2" style="width:80px;height:80px;background-color:green;"></div>
<br>
<div id="div3" style="width:80px;height:80px;background-color:blue;"></div>
</body>

</body>
</html>
```

jQuery fadeTo()

演示 jQuery fadeTo() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").fadeTo("slow",0.15);
    $("#div2").fadeTo("slow",0.4);
    $("#div3").fadeTo("slow",0.7);
  });
});
</script>
</head>

<body>

<p>演示带有不同参数的 fadeTo() 方法。</p>
<button>点击这里，使三个矩形淡出</button>
<br><br>
<div id="div1" style="width:80px;height:80px;background-color:red;"></div>
<br>
<div id="div2" style="width:80px;height:80px;background-color:green;"></div>
<br>
<div id="div3" style="width:80px;height:80px;background-color:blue;"></div>

</body>
</html>
```

jQuery Fading 方法

通过 jQuery，您可以实现元素的淡入淡出效果。

jQuery 拥有下面四种 fade 方法：

- fadeIn()
- fadeOut()
- fadeToggle()
- fadeTo()

jQuery fadeIn() 方法

jQuery fadeIn() 用于淡入已隐藏的元素。

语法：

```
$(selector).fadeIn(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是 fading 完成后所执行的函数名称。

下面的例子演示了带有不同参数的 fadeIn() 方法：

实例

```
$("button").click(function(){
  $("#div1").fadeIn();
  $("#div2").fadeIn("slow");
  $("#div3").fadeIn(3000);
});
```

jQuery fadeOut() 方法

jQuery fadeOut() 方法用于淡出可见元素。

语法：

```
$(selector).fadeOut(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是 fading 完成后所执行的函数名称。

下面的例子演示了带有不同参数的 fadeOut() 方法：

实例

```
$("button").click(function(){
  $("#div1").fadeOut();
  $("#div2").fadeOut("slow");
  $("#div3").fadeOut(3000);
});
```

jQuery fadeToggle() 方法

jQuery fadeToggle() 方法可以在 fadeIn() 与 fadeOut() 方法之间进行切换。

如果元素已淡出，则 fadeToggle() 会向元素添加淡入效果。

如果元素已淡入，则 fadeToggle() 会向元素添加淡出效果。

语法：

```
$(selector).fadeToggle(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是 fading 完成后所执行的函数名称。

下面的例子演示了带有不同参数的 fadeToggle() 方法：

实例

```
$("button").click(function(){
  $("#div1").fadeToggle();
  $("#div2").fadeToggle("slow");
  $("#div3").fadeToggle(3000);
});
```

jQuery fadeTo() 方法

jQuery fadeTo() 方法允许渐变为给定的不透明度（值介于 0 与 1 之间）。

语法：

```
$(selector).fadeTo(speed,opacity,callback);
```

必需的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

fadeTo() 方法中必需的 opacity 参数将淡入淡出效果设置为给定的不透明度（值介于 0 与 1 之间）。

可选的 callback 参数是该函数完成后所执行的函数名称。

下面的例子演示了带有不同参数的 fadeTo() 方法：

实例

```
$("button").click(function(){
  $("#div1").fadeTo("slow",0.15);
  $("#div2").fadeTo("slow",0.4);
  $("#div3").fadeTo("slow",0.7);
});
```

jQuery 效果参考手册

如需全面查阅 jQuery 效果，请访问我们的 jQuery 效果参考手册。

jQuery 效果 - 滑动

jQuery 滑动方法可使元素上下滑动。

实例

jQuery slideDown()

演示 jQuery slideDown() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $(".flip").click(function(){
    $(".panel").slideDown("slow");
  });
});
</script>

<style type="text/css">
div.panel,p.flip
{
margin:0px;
padding:5px;
text-align:center;
background:#e5eecc;
border:solid 1px #c3c3c3;
}
div.panel
{
height:120px;
display:none;
}
</style>
</head>

<body>

<div class="panel">
<p>W3School - 领先的 Web 技术教程站点</p>
<p>在 W3School，你可以找到你所需要的所有网站建设教程。</p>
</div>

<p class="flip">请点击这里</p>

</body>
</html>
```

jQuery slideUp()

演示 jQuery slideUp() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script type="text/javascript">
$(document).ready(function(){
  $(".flip").click(function(){
    $(".panel").slideUp("slow");
  });
});
</script>

<style type="text/css">
div.panel,p.flip
{
margin:0px;
padding:5px;
text-align:center;
background:#e5eecc;
border:solid 1px #c3c3c3;
}
div.panel
{
height:120px;
}
</style>
</head>

<body>

<div class="panel">
<p>W3School - 领先的 Web 技术教程站点</p>
<p>在 W3School，你可以找到你所需要的所有网站建设教程。</p>
</div>

<p class="flip">请点击这里</p>

</body>
</html>
```

jQuery slideToggle()

演示 jQuery slideToggle() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script type="text/javascript">
$(document).ready(function(){
$(".flip").click(function(){
    $(".panel").slideToggle("slow");
  });
});
</script>

<style type="text/css">
div.panel,p.flip
{
margin:0px;
padding:5px;
text-align:center;
background:#e5eecc;
border:solid 1px #c3c3c3;
}
div.panel
{
height:120px;
display:none;
}
</style>
</head>

<body>

<div class="panel">
<p>W3School - 领先的 Web 技术教程站点</p>
<p>在 W3School，你可以找到你所需要的所有网站建设教程。</p>
</div>

<p class="flip">请点击这里</p>

</body>
</html>
```

jQuery 滑动方法

通过 jQuery，您可以在元素上创建滑动效果。

jQuery 拥有以下滑动方法：

- slideDown()
- slideUp()
- slideToggle()

jQuery slideDown() 方法

jQuery slideDown() 方法用于向下滑动元素。

语法：

```
$(selector).slideDown(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是滑动完成后所执行的函数名称。

下面的例子演示了 slideDown() 方法：

实例

```
$("#flip").click(function(){
  $("#panel").slideDown();
});
```

jQuery slideUp() 方法

jQuery slideUp() 方法用于向上滑动元素。

语法：

```
$(selector).slideUp(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是滑动完成后所执行的函数名称。

下面的例子演示了 slideUp() 方法：

实例

```
$("#flip").click(function(){
  $("#panel").slideUp();
});
```

jQuery slideToggle() 方法

jQuery slideToggle() 方法可以在 slideDown() 与 slideUp() 方法之间进行切换。

如果元素向下滑动，则 slideToggle() 可向上滑动它们。

如果元素向上滑动，则 slideToggle() 可向下滑动它们。

```
$(selector).slideToggle(speed,callback);
```

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是滑动完成后所执行的函数名称。

下面的例子演示了 slideToggle() 方法：

实例

```
$("#flip").click(function(){
  $("#panel").slideToggle();
});
```

jQuery 效果参考手册

如需全面查阅 jQuery 效果，请访问我们的 jQuery 效果参考手册。

jQuery 效果 - 动画

jQuery animate() 方法允许您创建自定义的动画。

jQuery 动画 - animate() 方法

jQuery animate() 方法用于创建自定义动画。

语法：

```
$(selector).animate({params},speed,callback);
```

必需的 params 参数定义形成动画的 CSS 属性。

可选的 speed 参数规定效果的时长。它可以取以下值："slow"、"fast" 或毫秒。

可选的 callback 参数是动画完成后所执行的函数名称。

下面的例子演示 animate() 方法的简单应用；它把 <div> 元素移动到左边，直到 left 属性等于 250 像素为止：

实例

```
$("button").click(function(){
  $("div").animate({left:'250px'});
});
```

提示：默认地，所有 HTML 元素都有一个静态位置，且无法移动。

如需对位置进行操作，要记得首先把元素的 CSS position 属性设置为 relative、fixed 或 absolute！

jQuery animate() - 操作多个属性

请注意，生成动画的过程中可同时使用多个属性：

实例

```
$("button").click(function(){
  $("div").animate({
    left:'250px',
    opacity:'0.5',
    height:'150px',
    width:'150px'
  });
});
```

提示：可以用 animate() 方法来操作所有 CSS 属性吗？

是的，几乎可以！不过，需要记住一件重要的事情：当使用 animate() 时，必须使用 Camel 标记法书写所有的属性名，比如，必须使用 paddingLeft 而不是 padding-left，使用 marginRight 而不是 margin-right，等等。

同时，色彩动画并不包含在核心 jQuery 库中。

如果需要生成颜色动画，您需要从 jQuery.com 下载 Color Animations 插件。

jQuery animate() - 使用相对值

也可以定义相对值（该值相对于元素的当前值）。需要在值的前面加上 += 或 -=：

实例

```
$("button").click(function(){
  $("div").animate({
    left:'250px',
    height:'+=150px',
    width:'+=150px'
  });
});
```

jQuery animate() - 使用预定义的值

您甚至可以把属性的动画值设置为 "show"、"hide" 或 "toggle"：

实例

```
$("button").click(function(){
  $("div").animate({
    height:'toggle'
  });
});
```

jQuery animate() - 使用队列功能

默认地，jQuery 提供针对动画的队列功能。

这意味着如果您在彼此之后编写多个 animate() 调用，jQuery 会创建包含这些方法调用的“内部”队列。然后逐一运行这些 animate 调用。

实例 1

隐藏，如果您希望在彼此之后执行不同的动画，那么我们要利用队列功能：

```
$("button").click(function(){
  var div=$("div");
  div.animate({height:'300px',opacity:'0.4'},"slow");
  div.animate({width:'300px',opacity:'0.8'},"slow");
  div.animate({height:'100px',opacity:'0.4'},"slow");
  div.animate({width:'100px',opacity:'0.8'},"slow");
});
```

实例 2

下面的例子把 <div> 元素移动到右边，然后增加文本的字号：

```
$("button").click(function(){
  var div=$("div");
  div.animate({left:'100px'},"slow");
  div.animate({fontSize:'3em'},"slow");
});
```

jQuery 停止动画

jQuery stop() 方法用于在动画或效果完成前对它们进行停止。

实例

jQuery stop() 滑动

演示 jQuery stop() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js"></script>
<script>
$(document).ready(function(){
  $("#flip").click(function(){
    $("#panel").slideDown(5000);
  });
  $("#stop").click(function(){
    $("#panel").stop();
  });
});
</script>

<style type="text/css">
#panel,#flip
{
padding:5px;
text-align:center;
background-color:#e5eecc;
border:solid 1px #c3c3c3;
}
#panel
{
padding:50px;
display:none;
}
</style>
</head>

<body>

<button id="stop">停止滑动</button>
<div id="flip">点击这里，向下滑动面板</div>
<div id="panel">Hello world!</div>

</body>
</html>
```

jQuery stop() 动画（带有参数）

演示 jQuery stop() 方法。

```
<!DOCTYPE html>
<html>
<head>
<script src="/jquery/jquery-1.11.1.min.js">
</script>
<script>
$(document).ready(function(){
  $("#start").click(function(){
    $("div").animate({left:'100px'},5000);
    $("div").animate({fontSize:'3em'},5000);
  });

  $("#stop").click(function(){
    $("div").stop();
  });

  $("#stop2").click(function(){
    $("div").stop(true);
  });

  $("#stop3").click(function(){
    $("div").stop(true,true);
  });

});
</script>
</head>
<body>

<button id="start">开始</button>
<button id="stop">停止</button>
<button id="stop2">停止所有</button>
<button id="stop3">停止但要完成</button>
<p><b>"开始"</b> 按钮会启动动画。</p>
<p><b>"停止"</b> 按钮会停止当前活动的动画，但允许已排队的动画向前执行。</p>
<p><b>"停止所有"</b> 按钮停止当前活动的动画，并清空动画队列；因此元素上的所有动画都会停止。</p>
<p><b>"停止但要完成"</b> 会立即完成当前活动的动画，然后停下来。</p>

<div style="background:#98bf21;height:100px;width:200px;position:absolute;">HELLO</div>

</body>
</html>
```

jQuery stop() 方法

jQuery stop() 方法用于停止动画或效果，在它们完成之前。

stop() 方法适用于所有 jQuery 效果函数，包括滑动、淡入淡出和自定义动画。

语法

```
$(selector).stop(stopAll,goToEnd);
```

可选的 stopAll 参数规定是否应该清除动画队列。默认是 false，即仅停止活动的动画，允许任何排入队列的动画向后执行。

可选的 goToEnd 参数规定是否立即完成当前动画。默认是 false。

因此，默认地，stop() 会清除在被选元素上指定的当前动画。

下面的例子演示 stop() 方法，不带参数：

实例

```
$("#stop").click(function(){
  $("#panel").stop();
});
```

jQuery 效果参考手册

如需全面查阅 jQuery 效果，请访问我们的 jQuery 效果参考手册。

jQuery Callback 函数

Callback 函数在当前动画 **100%** 完成之后执行。

jQuery 动画的问题

许多 jQuery 函数涉及动画。这些函数也许会将 *speed* 或 *duration* 作为可选参数。

例子：*$("p").hide("slow")*

speed 或 *duration* 参数可以设置许多不同的值，比如 "slow", "fast", "normal" 或毫秒。

实例

```
$("button").click(function(){
$("p").hide(1000);
});
```

由于 JavaScript 语句（指令）是逐一执行的 - 按照次序，动画之后的语句可能会产生错误或页面冲突，因为动画还没有完成。

为了避免这个情况，您可以以参数的形式添加 Callback 函数。

jQuery Callback 函数

当动画 100% 完成后，即调用 Callback 函数。

典型的语法：

```
$(selector).hide(speed,callback)
```

callback 参数是一个在 hide 操作完成后被执行的函数。

错误（没有 **callback**）

```
$("p").hide(1000);
alert("The paragraph is now hidden");
```

正确（有 **callback**）

```
$("p").hide(1000,function(){
alert("The paragraph is now hidden");
});
```

结论：如果您希望在一个涉及动画的函数之后来执行语句，请使用 callback 函数。

jQuery - Chaining

通过 **jQuery**，您可以把动作/方法链接起来。

Chaining 允许我们在一条语句中允许多个 **jQuery** 方法（在相同的元素上）。

jQuery 方法链接

直到现在，我们都是一次写一条 jQuery 语句（一条接着另一条）。

不过，有一种名为链接（chaining）的技术，允许我们在相同的元素上运行多条 jQuery 命令，一条接着另一条。

提示：这样的话，浏览器就不必多次查找相同的元素。

如需链接一个动作，您只需简单地把该动作追加到之前的动作上。

例子 1

下面的例子把 css(), slideUp(), and slideDown() 链接在一起。"p1" 元素首先会变为红色，然后向上滑动，然后向下滑动：

```
$("#p1").css("color","red").slideUp(2000).slideDown(2000);
```

如果需要，我们也可以添加多个方法调用。

提示：当进行链接时，代码行会变得很差。不过，jQuery 在语法上不是很严格；您可以按照希望的格式来写，包含折行和缩进。

例子 2

这样写也可以运行：

```
$("#p1").css("color","red")
  .slideUp(2000)
  .slideDown(2000);
```

jQuery 会抛掉多余的空格，并按照一行长代码来执行上面的代码行。

jQuery HTML

jQuery - 获得内容和属性

jQuery 拥有可操作 **HTML** 元素和属性的**强**大方法。

jQuery DOM 操作

jQuery 中非常重要的部分，就是操作 DOM 的能力。

jQuery 提供一系列与 DOM 相关的方法，这使访问和操作元素和属性变得很容易。

提示：DOM = Document Object Model（文档对象模型）

DOM 定义访问 HTML 和 XML 文档的标准：

“W3C 文档对象模型独立于平台和语言的界面，允许程序和脚本动态访问和更新文档的内容、结构以及样式。”

获得内容 - text()、html() 以及 val()

三个简单实用的用于 DOM 操作的 jQuery 方法：

- text() - 设置或返回所选元素的文本内容
- html() - 设置或返回所选元素的内容（包括 HTML 标记）
- val() - 设置或返回表单字段的值

下面的例子演示如何通过 jQuery text() 和 html() 方法来获得内容：

实例

```
$("#btn1").click(function(){
  alert("Text: " + $("#test").text());
});
$("#btn2").click(function(){
  alert("HTML: " + $("#test").html());
});
```

下面的例子演示如何通过 jQuery val() 方法获得输入字段的值：

实例

```
$("#btn1").click(function(){
  alert("Value: " + $("#test").val());
});
```

获取属性 - attr()

jQuery attr() 方法用于获取属性值。

下面的例子演示如何获得链接中 href 属性的值：

实例

```
$("button").click(function(){
  alert($("#w3s").attr("href"));
});
```

下一章会讲解如何设置（改变）内容和属性值。

jQuery HTML 参考手册

如需有关 jQuery HTML 方法的完整内容，请访问以下参考手册：

jQuery - 设置内容和属性

设置内容 - text()、html() 以及 val()

我们将使用前一章中的三个相同的方法来设置内容：

- text() - 设置或返回所选元素的文本内容
- html() - 设置或返回所选元素的内容（包括 HTML 标记）
- val() - 设置或返回表单字段的值

下面的例子演示如何通过 text()、html() 以及 val() 方法来设置内容：

实例

```
$("#btn1").click(function(){
  $("#test1").text("Hello world!");
});
$("#btn2").click(function(){
  $("#test2").html("<b>Hello world!</b>");
});
$("#btn3").click(function(){
  $("#test3").val("Dolly Duck");
});
```

text()、html() 以及 val() 的回调函数

上面的三个 jQuery 方法：text()、html() 以及 val()，同样拥有回调函数。回调函数由两个参数：被选元素列表中当前元素的下标，以及原始（旧的）值。然后以函数新值返回您希望使用的字符串。

下面的例子演示带有回调函数的 text() 和 html()：

实例

```
$("#btn1").click(function(){
  $("#test1").text(function(i,origText){
    return "Old text: " + origText + " New text: Hello world!
    (index: " + i + ")";
  });
});

$("#btn2").click(function(){
  $("#test2").html(function(i,origText){
    return "Old html: " + origText + " New html: Hello <b>world!</b>
    (index: " + i + ")";
  });
});
```

设置属性 - attr()

jQuery attr() 方法也用于设置/改变属性值。

下面的例子演示如何改变（设置）链接中 href 属性的值：

实例

```
$("button").click(function(){
  $("#w3s").attr("href","http://www.w3school.com.cn/jquery");
});
```

attr() 方法也允许您同时设置多个属性。

下面的例子演示如何同时设置 href 和 title 属性：

实例

```
$("button").click(function(){
  $("#w3s").attr({
    "href" : "http://www.w3school.com.cn/jquery",
    "title" : "W3School jQuery Tutorial"
  });
});
```

attr() 的回调函数

jQuery 方法 attr()，也提供回调函数。回调函数由两个参数：被选元素列表中当前元素的下标，以及原始（旧的）值。然后以函数新值返回您希望使用的字符串。

下面的例子演示带有回调函数的 attr() 方法：

实例

```
$("button").click(function(){
  $("#w3s").attr("href", function(i,origValue){
    return origValue + "/jquery";
  });
});
```

jQuery HTML 参考手册

如需有关 jQuery HTML 方法的完整内容，请访问以下参考手册：

jQuery - 添加元素

通过 **jQuery**，可以很容易地添加新元素/内容。

添加新的 HTML 内容

我们将学习用于添加新内容的四个 jQuery 方法：

- append() - 在被选元素的结尾插入内容
- prepend() - 在被选元素的开头插入内容
- after() - 在被选元素之后插入内容
- before() - 在被选元素之前插入内容

jQuery append() 方法

jQuery append() 方法在被选元素的结尾插入内容。

实例

```
$("p").append("Some appended text.");
```

jQuery prepend() 方法

jQuery prepend() 方法在被选元素的开头插入内容。

实例

```
$("p").prepend("Some prepended text.");
```

通过 append() 和 prepend() 方法添加若干新元素

在上面的例子中，我们只在被选元素的开头/结尾插入文本/HTML。

不过，append() 和 prepend() 方法能够通过参数接收无限数量的新元素。可以通过 jQuery 来生成文本/HTML（就像上面的例子那样），或者通过 JavaScript 代码和 DOM 元素。

在下面的例子中，我们创建若干个新元素。这些元素可以通过 text/HTML、jQuery 或者 JavaScript/DOM 来创建。然后我们通过 append() 方法把这些新元素追加到文本中（对 prepend() 同样有效）：

实例

```
function appendText()
{
var txt1="<p>Text.</p>";               // 以 HTML 创建新元素
var txt2=$("<p></p>").text("Text.");   // 以 jQuery 创建新元素
var txt3=document.createElement("p");  // 以 DOM 创建新元素
txt3.innerHTML="Text.";
$("p").append(txt1,txt2,txt3);         // 追加新元素
}
```

jQuery after() 和 before() 方法

jQuery after() 方法在被选元素之后插入内容。

jQuery before() 方法在被选元素之前插入内容。

实例

```
$("img").after("Some text after");

$("img").before("Some text before");
```

通过 after() 和 before() 方法添加若干新元素

after() 和 before() 方法能够通过参数接收无限数量的新元素。可以通过 text/HTML、jQuery 或者 JavaScript/DOM 来创建新元素。

在下面的例子中，我们创建若干新元素。这些元素可以通过 text/HTML、jQuery 或者 JavaScript/DOM 来创建。然后我们通过 after() 方法把这些新元素插到文本中（对 before() 同样有效）：

实例

```
function afterText()
{
var txt1="<b>I </b>";                    // 以 HTML 创建新元素
var txt2=$("<i></i>").text("love ");     // 通过 jQuery 创建新元素
var txt3=document.createElement("big");  // 通过 DOM 创建新元素
txt3.innerHTML="jQuery!";
$("img").after(txt1,txt2,txt3);          // 在 img 之后插入新元素
}
```

jQuery HTML 参考手册

如需有关 jQuery HTML 方法的完整内容，请访问以下参考手册：

jQuery - 删除元素

通过 **jQuery**，可以很容易地删除已有的 **HTML** 元素。

删除元素/内容

如需删除元素和内容，一般可使用以下两个 jQuery 方法：

- remove() - 删除被选元素（及其子元素）
- empty() - 从被选元素中删除子元素

jQuery remove() 方法

jQuery remove() 方法删除被选元素及其子元素。

实例

```
$("#div1").remove();
```

jQuery empty() 方法

jQuery empty() 方法删除被选元素的子元素。

实例

```
$("#div1").empty();
```

过滤被删除的元素

jQuery remove() 方法也可接受一个参数，允许您对被删元素进行过滤。

该参数可以是任何 jQuery 选择器的语法。

下面的例子删除 class="italic" 的所有 <p> 元素：

实例

```
$("p").remove(".italic");
```

jQuery HTML 参考手册

如需有关 jQuery HTML 方法的完整内容，请访问以下参考手册：

jQuery - 获取并设置 CSS 类

通过 **jQuery**，可以很容易地对 **CSS** 元素进行操作。

切换类

jQuery 操作 CSS

jQuery 拥有若干进行 CSS 操作的方法。我们将学习下面这些：

- addClass() - 向被选元素添加一个或多个类
- removeClass() - 从被选元素删除一个或多个类
- toggleClass() - 对被选元素进行添加/删除类的切换操作
- css() - 设置或返回样式属性

实例样式表

下面的样式表将用于本页的所有例子：

```
.important
{
font-weight:bold;
font-size:xx-large;
}

.blue
{
color:blue;
}
```

jQuery addClass() 方法

下面的例子展示如何向不同的元素添加 class 属性。当然，在添加类时，您也可以选取多个元素：

实例

```
$("button").click(function(){
  $("h1,h2,p").addClass("blue");
  $("div").addClass("important");
});
```

您也可以在 addClass() 方法中规定多个类：

实例

```
$("button").click(function(){
  $("#div1").addClass("important blue");
});
```

jQuery removeClass() 方法

下面的例子演示如何不同的元素中删除指定的 class 属性：

实例

```
$("button").click(function(){
  $("h1,h2,p").removeClass("blue");
});
```

jQuery toggleClass() 方法

下面的例子将展示如何使用 jQuery toggleClass() 方法。该方法对被选元素进行添加/删除类的切换操作：

实例

```
$("button").click(function(){
  $("h1,h2,p").toggleClass("blue");
});
```

jQuery css() 方法

我们将在下一章讲解 jQuery css() 方法。

jQuery HTML 参考手册

如需有关 jQuery CSS 方法的完整内容，请访问我们的 jQuery CSS 操作参考手册

jQuery - css() 方法

jQuery css() 方法

css() 方法设置或返回被选元素的一个或多个样式属性。

返回 CSS 属性

如需返回指定的 CSS 属性的值，请使用如下语法：

```
css("propertyname");
```

下面的例子将返回首个匹配元素的 background-color 值：

实例

```
$("p").css("background-color");
```

设置 CSS 属性

如需设置指定的 CSS 属性，请使用如下语法：

```
css("propertyname","value");
```

下面的例子将为所有匹配元素设置 background-color 值：

实例

```
$("p").css("background-color","yellow");
```

设置多个 CSS 属性

如需设置多个 CSS 属性，请使用如下语法：

```
css({"propertyname":"value","propertyname":"value",...});
```

下面的例子将为所有匹配元素设置 background-color 和 font-size：

实例

```
$("p").css({"background-color":"yellow","font-size":"200%"});
```

jQuery HTML 参考手册

如需有关 jQuery CSS 方法的完整内容，请访问我们的 jQuery CSS 操作参考手册

jQuery - 尺寸

通过 **jQuery**，很容易处理元素和浏览器窗口的尺寸。

jQuery 尺寸 方法

jQuery 提供多个处理尺寸的重要方法：

- width()
- height()
- innerWidth()
- innerHeight()
- outerWidth()
- outerHeight()

jQuery width() 和 height() 方法

width() 方法设置或返回元素的宽度（不包括内边距、边框或外边距）。

height() 方法设置或返回元素的高度（不包括内边距、边框或外边距）。

下面的例子返回指定的 <div> 元素的宽度和高度：

实例

```
$("button").click(function(){
  var txt="";
  txt+="Width: " + $("#div1").width() + "</br>";
  txt+="Height: " + $("#div1").height();
  $("#div1").html(txt);
});
```

jQuery innerWidth() 和 innerHeight() 方法

innerWidth() 方法返回元素的宽度（包括内边距）。

innerHeight() 方法返回元素的高度（包括内边距）。

下面的例子返回指定的 <div> 元素的 inner-width/height：

实例

```
$("button").click(function(){
  var txt="";
  txt+="Inner width: " + $("#div1").innerWidth() + "</br>";
  txt+="Inner height: " + $("#div1").innerHeight();
  $("#div1").html(txt);
});
```

jQuery outerWidth() 和 outerHeight() 方法

outerWidth() 方法返回元素的宽度（包括内边距和边框）。

outerHeight() 方法返回元素的高度（包括内边距和边框）。

下面的例子返回指定的 <div> 元素的 outer-width/height：

实例

```
$("button").click(function(){
  var txt="";
  txt+="Outer width: " + $("#div1").outerWidth() + "</br>";
  txt+="Outer height: " + $("#div1").outerHeight();
  $("#div1").html(txt);
});
```

outerWidth(true) 方法返回元素的宽度（包括内边距、边框和外边距）。

outerHeight(true) 方法返回元素的高度（包括内边距、边框和外边距）。

实例

```
$("button").click(function(){
  var txt="";
  txt+="Outer width (+margin): " + $("#div1").outerWidth(true) + "</br>";
  txt+="Outer height (+margin): " + $("#div1").outerHeight(true);
  $("#div1").html(txt);
});
```

jQuery - 更多的 width() 和 height()

下面的例子返回文档（HTML 文档）和窗口（浏览器视口）的宽度和高度：

实例

```
$("button").click(function(){
  var txt="";
  txt+="Document width/height: " + $(document).width();
  txt+="x" + $(document).height() + "\n";
  txt+="Window width/height: " + $(window).width();
  txt+="x" + $(window).height();
  alert(txt);
});
```

下面的例子设置指定的 <div> 元素的宽度和高度：

实例

```
$("button").click(function(){
  $("#div1").width(500).height(500);
});
```

jQuery CSS 参考手册

如需关于 jQuery Dimensions 的完整参考，请访问我们的 jQuery 尺寸参考手册。

jQuery 遍历

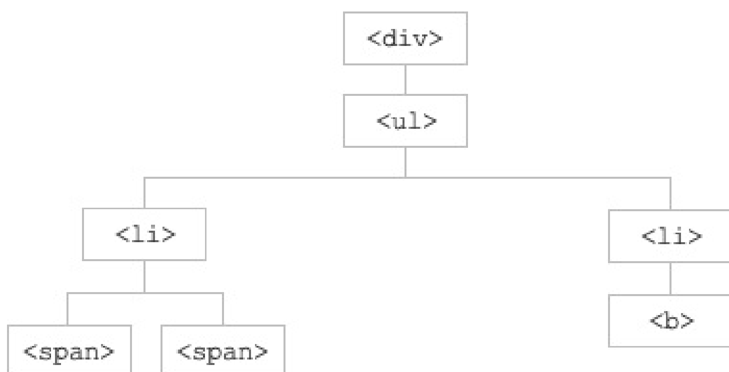
jQuery 遍历

什么是遍历？

jQuery 遍历，意为“移动”，用于根据其相对于其他元素的关系来“查找”（或选取）HTML 元素。以某项选择开始，并沿着这个选择移动，直到抵达您期望的元素为止。

下图展示了一个家族树。通过 jQuery 遍历，您能够从被选（当前的）元素开始，轻松地在家族树中向上移动（祖先），向下移动（子孙），水平移动（同胞）。这种移动被称为对 DOM 进行遍历。

图示解释：



- `<div>` 元素是 `<ul>` 的父元素，同时是其中所有内容的祖先。
- `<ul>` 元素是 `<li>` 元素的父元素，同时是 `<div>` 的子元素
- 左边的 `<li>` 元素是 `<span>` 的父元素， `<ul>` 的子元素，同时是 `<div>` 的后代。
- `<span>` 元素是 `<li>` 的子元素，同时是 `<ul>` 和 `<div>` 的后代。
- 两个 `<li>` 元素是同胞（拥有相同的父元素）。
- 右边的 `<li>` 元素是 `<b>` 的父元素， `<ul>` 的子元素，同时是 `<div>` 的后代。
- `<b>` 元素是右边的 `<li>` 的子元素，同时是 `<ul>` 和 `<div>` 的后代。

提示：祖先是父、祖父、曾祖父等等。后代是子、孙、曾孙等等。同胞拥有相同的父。

遍历 DOM

jQuery 提供了多种遍历 DOM 的方法。

遍历方法中最大的种类是树遍历（tree-traversal）。

下一章会讲解如何在 DOM 树中向上、下以及同级移动。

jQuery 遍历参考手册

如需了解所有的 jQuery 遍历方法，请访问我们的 jQuery 遍历参考手册。

jQuery 遍历 - 祖先

祖先是父、祖父或曾祖父等等。

通过 jQuery，您能够向上遍历 DOM 树，以查找元素的祖先。

向上遍历 DOM 树

这些 jQuery 方法很有用，它们用于向上遍历 DOM 树：

- parent()
- parents()
- parentsUntil()

jQuery parent() 方法

parent() 方法返回被选元素的直接父元素。

该方法只会向上一级对 DOM 树进行遍历。

下面的例子返回每个 <span> 元素的的直接父元素：

实例

```
$(document).ready(function(){
  $("span").parent();
});
```

jQuery parents() 方法

parents() 方法返回被选元素的所有祖先元素，它一路向上直到文档的根元素 (<html>)。

下面的例子返回所有 <span> 元素的所有祖先：

实例

```
$(document).ready(function(){
  $("span").parents();
});
```

您也可以使用可选参数来过滤对祖先元素的搜索。

下面的例子返回所有 <span> 元素的所有祖先，并且它是 <ul> 元素：

实例

```
$(document).ready(function(){
  $("span").parents("ul");
});
```

jQuery parentsUntil() 方法

parentsUntil() 方法返回介于两个给定元素之间的所有祖先元素。

下面的例子返回介于 <span> 与 <div> 元素之间的所有祖先元素：

实例

```
$(document).ready(function(){
  $("span").parentsUntil("div");
});
```

jQuery 遍历参考手册

如需了解所有的 jQuery 遍历方法，请访问我们的 jQuery 遍历参考手册。

jQuery 遍历 - 后代

后代是子、孙、曾孙等等。

通过 jQuery，您能够向下遍历 DOM 树，以查找元素的后代。

向下遍历 DOM 树

下面是两个用于向下遍历 DOM 树的 jQuery 方法：

- children()
- find()

jQuery children() 方法

children() 方法返回被选元素的所有直接子元素。

该方法只会向下一级对 DOM 树进行遍历。

下面的例子返回每个 <div> 元素的所有直接子元素：

实例

```
$(document).ready(function(){
  $("div").children();
});
```

您也可以使用可选参数来过滤对子元素的搜索。

下面的例子返回类名为 "1" 的所有 <p> 元素，并且它们是 <div> 的直接子元素：

实例

```
$(document).ready(function(){
  $("div").children("p.1");
});
```

jQuery find() 方法

find() 方法返回被选元素的后代元素，一路向下直到最后一个后代。

下面的例子返回属于 <div> 后代的所有 <span> 元素：

实例

```
$(document).ready(function(){
  $("div").find("span");
});
```

下面的例子返回 <div> 的所有后代：

实例

```
$(document).ready(function(){
  $("div").find("*");
});
```

jQuery 遍历参考手册

如需了解所有的 jQuery 遍历方法，请访问我们的 jQuery 遍历参考手册。

jQuery 遍历 - 同胞

同胞拥有相同的父元素。

通过 jQuery，您能够在 DOM 树中遍历元素的同胞元素。

在 DOM 树中水平遍历

有许多有用的方法让我们在 DOM 树进行水平遍历：

- siblings()
- next()
- nextAll()
- nextUntil()
- prev()
- prevAll()
- prevUntil()

jQuery siblings() 方法

siblings() 方法返回被选元素的所有同胞元素。

下面的例子返回 <h2> 的所有同胞元素：

实例

```
$(document).ready(function(){
  $("h2").siblings();
});
```

您也可以使用可选参数来过滤对同胞元素的搜索。

下面的例子返回属于 <h2> 的同胞元素的所有 <p> 元素：

实例

```
$(document).ready(function(){
  $("h2").siblings("p");
});
```

jQuery next() 方法

next() 方法返回被选元素的下一个同胞元素。

该方法只返回一个元素。

下面的例子返回 <h2> 的下一个同胞元素：

实例

```
$(document).ready(function(){
  $("h2").next();
});
```

jQuery nextAll() 方法

nextAll() 方法返回被选元素的所有跟随的同胞元素。

下面的例子返回 <h2> 的所有跟随的同胞元素：

实例

```
$(document).ready(function(){
  $("h2").nextAll();
});
```

jQuery nextUntil() 方法

nextUntil() 方法返回介于两个给定参数之间的所有跟随的同胞元素。

下面的例子返回介于 <h2> 与 <h6> 元素之间的所有同胞元素：

实例

```
$(document).ready(function(){
  $("h2").nextUntil("h6");
});
```

jQuery prev(), prevAll() & prevUntil() 方法

prev(), prevAll() 以及 prevUntil() 方法的工作方式与上面的方法类似，只不过方向相反而已：它们返回的是前面的同胞元素（在 DOM 树中沿着同胞元素向后遍历，而不是向前）。

jQuery 遍历参考手册

如需了解所有的 jQuery 遍历方法，请访问我们的 jQuery 遍历参考手册。

jQuery 遍历 - 过滤

缩写搜索元素的范围

三个最基本的过滤方法是：first(), last() 和 eq()，它们允许您基于其在一组元素中的位置来选择一个特定的元素。

其他过滤方法，比如 filter() 和 not() 允许您选取匹配或不匹配某项指定标准的元素。

jQuery first() 方法

first() 方法返回被选元素的首个元素。

下面的例子选取首个 <div> 元素内部的第一个 <p> 元素：

实例

```
$(document).ready(function(){
  $("div p").first();
});
```

jQuery last() 方法

last() 方法返回被选元素的最后一个元素。

下面的例子选择最后一个 <div> 元素中的最后一个 <p> 元素：

实例

```
$(document).ready(function(){
  $("div p").last();
});
```

jQuery eq() 方法

eq() 方法返回被选元素中带有指定索引号的元素。

索引号从 0 开始，因此首个元素的索引号是 0 而不是 1。下面的例子选取第二个 <p> 元素（索引号 1）：

实例

```
$(document).ready(function(){
  $("p").eq(1);
});
```

jQuery filter() 方法

filter() 方法允许您规定一个标准。不匹配这个标准的元素会被从集合中删除，匹配的元素会被返回。

下面的例子返回带有类名 "intro" 的所有 <p> 元素：

实例

```
$(document).ready(function(){
  $("p").filter(".intro");
});
```

jQuery not() 方法

not() 方法返回不匹配标准的所有元素。

提示：not() 方法与 filter() 相反。

下面的例子返回不带有类名 "intro" 的所有 <p> 元素：

实例

```
$(document).ready(function(){
  $("p").not(".intro");
});
```

jQuery 遍历参考手册

如需了解所有的 jQuery 遍历方法，请访问我们的 jQuery 遍历参考手册。

jQuery - AJAX

jQuery - AJAX 简介

AJAX 是与服务器交换数据的艺术，它在不重载全部页面的情况下，实现了对部分网页的更新。

jQuery AJAX 实例



```
<script>$(document).ready(function(){ $("#btn1").click(function(){ $('#test').load('/exam
<h3>请点击下面的按钮，通过 jQuery AJAX 改变这段文本。</h3>
<button id="btn1" type="button">获得外部的内容</button>
```

什么是 AJAX？

AJAX = 异步 JavaScript 和 XML（Asynchronous JavaScript and XML）。

简短地说，在不重载整个网页的情况下，AJAX 通过后台加载数据，并在网页上进行显示。

使用 AJAX 的应用程序案例：谷歌地图、腾讯微博、优酷视频、人人网等等。

您可以在我们的 AJAX 教程中学到更多有关 AJAX 的知识。

关于 jQuery 与 AJAX

jQuery 提供多个与 AJAX 有关的方法。

通过 jQuery AJAX 方法，您能够使用 HTTP Get 和 HTTP Post 从远程服务器上请求文本、HTML、XML 或 JSON - 同时您能够把这些外部数据直接载入网页的被选元素中。

提示：如果没有 jQuery，AJAX 编程还是有些难度的。

编写常规的 AJAX 代码并不容易，因为不同的浏览器对 AJAX 的实现并不相同。这意味着您必须编写额外的代码对浏览器进行测试。不过，jQuery 团队为我们解决了这个难题，我们只需要一行简单的代码，就可以实现 AJAX 功能。

jQuery AJAX 方法

在下面的章节，我们将学习到最重要的 jQuery AJAX 方法。

jQuery - AJAX load() 方法

jQuery load() 方法

jQuery load() 方法是简单但强大的 AJAX 方法。

load() 方法从服务器加载数据，并把返回的数据放入被选元素中。

语法：

```
$(_selector_).load(_URL_,_data_,_callback_);
```

必需的 *URL* 参数规定您希望加载的 URL。

可选的 *data* 参数规定与请求一同发送的查询字符串键/值对集合。

可选的 *callback* 参数是 load() 方法完成后所执行的函数名称。

这是示例文件（"demo_test.txt"）的内容：

```
<h2>jQuery and AJAX is FUN!!!</h2>
<p id="p1">This is some text in a paragraph.</p>
```

下面的例子会把文件 "demo_test.txt" 的内容加载到指定的 <div> 元素中：

示例

```
$("#div1").load("demo_test.txt");
```

也可以把 jQuery 选择器添加到 URL 参数。

下面的例子把 "demo_test.txt" 文件中 id="p1" 的元素的内容，加载到指定的 <div> 元素中：

实例

```
$("#div1").load("demo_test.txt #p1");
```

可选的 callback 参数规定当 load() 方法完成后所要允许的回调函数。回调函数可以设置不同的参数：

- *responseTxt* - 包含调用成功时的结果内容
- *statusTXT* - 包含调用的状态
- *xhr* - 包含 XMLHttpRequest 对象

下面的例子会在 load() 方法完成后显示一个提示框。如果 load() 方法已成功，则显示“外部内容加载成功！”，而如果失败，则显示错误消息：

实例

```
$("button").click(function(){
  $("#div1").load("demo_test.txt",function(responseTxt,statusTxt,xhr){
    if(statusTxt=="success")
      alert("外部内容加载成功！");
    if(statusTxt=="error")
      alert("Error: "+xhr.status+": "+xhr.statusText);
  });
});
```

jQuery AJAX 参考手册

如需完整的 AJAX 方法参考，请访问我们的 jQuery AJAX 参考手册。

jQuery - AJAX get() 和 post() 方法

jQuery get() 和 post() 方法用于通过 HTTP GET 或 POST 请求从服务器请求数据。

HTTP 请求：GET vs. POST

两种在客户端和服务器端进行请求-响应的常用方法是：GET 和 POST。

- *GET* - 从指定的资源请求数据
- *POST* - 向指定的资源提交要处理的数据

GET 基本上用于从服务器获得（取回）数据。注释：GET 方法可能返回缓存数据。

POST 也可用于从服务器获取数据。不过，POST 方法不会缓存数据，并且常用于连同请求一起发送数据。

如需学习更多有关 GET 和 POST 以及两方法差异的知识，请阅读我们的 HTTP 方法 - GET 对比 POST。

jQuery $.get() 方法

$.get() 方法通过 HTTP GET 请求从服务器上请求数据。

语法：

```
$.get(_URL_,_callback_);
```

必需的 *URL* 参数规定您希望请求的 URL。

可选的 *callback* 参数是请求成功后所执行的函数名。

下面的例子使用 $.get() 方法从服务器上的一个文件中取回数据：

实例

```
$("button").click(function(){
  $.get("demo_test.asp",function(data,status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```

$.get() 的第一个参数是我们希望请求的 URL（"demo_test.asp"）。

第二个参数是回调函数。第一个回调参数存有被请求页面的内容，第二个回调参数存有请求的状态。

提示：这个 ASP 文件 ("demo_test.asp") 类似这样：

```
<%
response.write("This is some text from an external ASP file.")
%>
```

jQuery $.post() 方法

$.post() 方法通过 HTTP POST 请求从服务器上请求数据。

语法：

```
$.post(_URL_,_data_,_callback_);
```

必需的 *URL* 参数规定您希望请求的 URL。

可选的 *data* 参数规定连同请求发送的数据。

可选的 *callback* 参数是请求成功后所执行的函数名。

下面的例子使用 $.post() 连同请求一起发送数据：

实例

```
$("button").click(function(){
  $.post("demo_test_post.asp",
  {
    name:"Donald Duck",
    city:"Duckburg"
  },
  function(data,status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```

$.post() 的第一个参数是我们希望请求的 URL ("demo_test_post.asp")。

然后我们连同请求（name 和 city）一起发送数据。

"demo_test_post.asp" 中的 ASP 脚本读取这些参数，对它们进行处理，然后返回结果。

第三个参数是回调函数。第一个回调参数存有被请求页面的内容，而第二个参数存有请求的状态。

提示：这个 ASP 文件 ("demo_test_post.asp") 类似这样：

```
<%
dim fname,city
fname=Request.Form("name")
city=Request.Form("city")
Response.Write("Dear " & fname & ". ")
Response.Write("Hope you live well in " & city & ".")
%>
```

jQuery AJAX 参考手册

如需完整的 AJAX 方法参考，请访问我们的 jQuery AJAX 参考手册。

jQuery 杂项

jQuery - noConflict() 方法

如何在页面上同时使用 **jQuery** 和其他框架？

jQuery 和其他 JavaScript 框架

正如您已经了解到的，jQuery 使用 $ 符号作为 jQuery 的简写。

如果其他 JavaScript 框架也使用 $ 符号作为简写怎么办？

其他一些 JavaScript 框架包括：MooTools、Backbone、Sammy、Cappuccino、Knockout、JavaScript MVC、Google Web Toolkit、Google Closure、Ember、Batman 以及 Ext JS。

其中某些框架也使用 $ 符号作为简写（就像 jQuery），如果您在用的两种不同的框架正在使用相同的简写符号，有可能导致脚本停止运行。

jQuery 的团队考虑到了这个问题，并实现了 noConflict() 方法。

jQuery noConflict() 方法

noConflict() 方法会释放会 $ 标识符的控制，这样其他脚本就可以使用它了。

实例

当然，您仍然可以通过全名替代简写的方式来使用 jQuery：

```
$.noConflict();
jQuery(document).ready(function(){
  jQuery("button").click(function(){
    jQuery("p").text("jQuery 仍在运行！");
  });
});
```

实例

您也可以创建自己的简写。noConflict() 可返回对 jQuery 的引用，您可以把它存入变量，以供稍后使用。请看这个例子：

```
var jq = $.noConflict();
jq(document).ready(function(){
  jq("button").click(function(){
    jq("p").text("jQuery 仍在运行！");
  });
});
```

实例

如果你的 jQuery 代码块使用 $ 简写，并且您不愿意改变这个快捷方式，那么您可以把 $ 符号作为变量传递给 ready 方法。这样就可以在函数内使用 $ 符号了 - 而在函数外，依旧不得不使用 "jQuery"：

```
$.noConflict();
jQuery(document).ready(function($){
  $("button").click(function(){
    $("p").text("jQuery 仍在运行！");
  });
});
```

jQuery 核心参考手册

如需完整的 jQuery 核心方法概览，请访问我们的 jQuery 核心参考手册。

JavaScript 高级教程

JavaScript 的历史

为了发挥 **JavaScript** 的全部潜力，了解它的本质、历史及局限性是十分重要的。

本节为您讲解 **JavaScript** 和客户端脚本的起源。

Nombas 和 ScriptEase

大概在 1992 年，一家称作 Nombas 的公司开发了一种叫做 C 减减（C-minus-minus，简称 Cmm）的嵌入式脚本语言。Cmm 背后的理念很简单：一个足够强大可以替代宏操作（macro）的脚本语言，同时保持与 C （和 C ++）足够的相似性，以便开发人员能很快学会。这个脚本语言捆绑在一个叫做 CEnvi 的共享软件中，它首次向开发人员展示了这种语言的威力。

Nombas 最终把 Cmm 的名字改成了 ScriptEase，原因是后面的部分（mm）听起来过于消极，同时字母 C “令人害怕”。

现在 ScriptEase 已经成为了 Nombas 产品背后的主要驱动力。

Netscape 发明了 JavaScript

当 Netscape Navigator 崭露头角时，Nombas 开发了一个可以嵌入网页中的 CEnvi 的版本。这些早期的试验被称为 Espresso Page（浓咖啡般的页面），它们代表了第一个在万维网上使用的客户端语言。而 Nombas 丝毫没有料到它的理念将会成为万维网的一块重要基石。

当网上冲浪越来越流行时，对于开发客户端脚本的需求也逐渐增大。此时，大部分因特网用户还仅仅通过 28.8 kbit/s 的调制解调器连接到网络，即便这时网页已经不断地变得更大和更复杂。而更加加剧用户痛苦的是，仅仅为了简单的表单有效性验证，就要与服务器进行多次地往返交互。设想一下，用户填完一个表单，点击提交按钮，等待了 30 秒的处理后，看到的却是一条告诉你忘记填写一个必要的字段。

那时正处于技术革新最前沿的 Netscape，开始认真考虑开发一种客户端脚本语言来解决简单的处理问题。

当时工作于 Netscape 的 Brendan Eich，开始着手为即将在 1995 年发行的 Netscape Navigator 2.0 开发一个称之为 LiveScript 的脚本语言，当时的目的是在浏览器和服务器（本来要叫它 LiveWire）端使用它。Netscape 与 Sun 及时完成 LiveScript 实现。

就在 Netscape Navigator 2.0 即将正式发布前，Netscape 将其更名为 JavaScript，目的是为了利用 Java 这个因特网时髦词汇。Netscape 的赌注最终得到回报，JavaScript 从此变成了因特网的必备组件。

三足鼎立

因为 JavaScript 1.0 如此成功，Netscape 在 Netscape Navigator 3.0 中发布了 1.1 版。恰巧那个时候，微软决定进军浏览器，发布了 IE 3.0 并搭载了一个 JavaScript 的克隆版，叫做 JScript（这样命名是为了避免与 Netscape 潜在的许可纠纷）。微软步入 Web 浏览器领域的这重要一步虽然令其声名狼藉，但也成为 JavaScript 语言发展过程中的重要一步。

在微软进入后，有 3 种不同的 JavaScript 版本同时存在：Netscape Navigator 3.0 中的 JavaScript、IE 中的 JScript 以及 CEnvi 中的 ScriptEase。与 C 和其他编程语言不同的是，JavaScript 并没有一个标准来统一其语法或特性，而这 3 种不同的版本恰恰突出了这个问题。随着业界担心的增加，这个语言的标准化显然已经势在必行。

标准化

1997 年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第 39 技术委员会（TC39）被委派来“标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语义”(http://www.ecma-international.org/memento/TC39.htm)。由来自 Netscape、Sun、微软、Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39 锤炼出了 ECMA-262，该标准定义了名为 ECMAScript 的全新脚本语言。

在接下来的几年里，国际标准化组织及国际电工委员会（ISO/IEC）也采纳 ECMAScript 作为标准（ISO/IEC-16262）。从此，Web 浏览器就开始努力（虽然有着不同的程度的成功和失败）将 ECMAScript 作为 JavaScript 实现的基础。

JavaScript 实现

JavaScript 的核心 **ECMAScript** 描述了该语言的语法和基本对象；

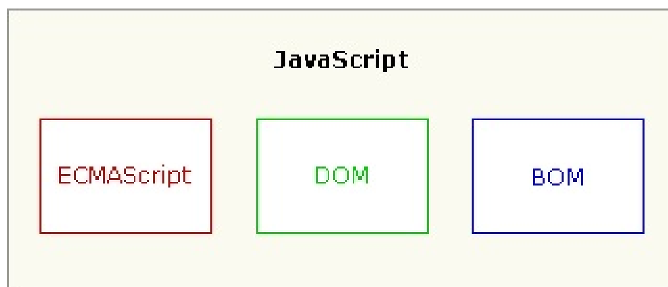
DOM 描述了处理网页内容的方法和接口；

BOM 描述了与浏览器进行交互的方法和接口。

ECMAScript、DOM 和 BOM

尽管 ECMAScript 是一个重要的标准，但它并不是 JavaScript 唯一的部分，当然，也不是唯一被标准化的部分。实际上，一个完整的 JavaScript 实现是由以下 3 个不同部分组成的：

- 核心（ECMAScript）
- 文档对象模型（DOM）
- 浏览器对象模型（BOM）



ECMAScript

ECMAScript 并不与任何具体浏览器相绑定，实际上，它也没有提到用于任何用户输入输出的方法（这点与 C 这类语言不同，它需要依赖外部的库来完成这类任务）。那么什么才是 ECMAScript 呢？ECMA-262 标准（第 2 段）的描述如下：

> "ECMAScript 可以为不同种类的宿主环境提供核心的脚本编程能力，因此核心的脚本语言是与任何特定的宿主环境分开进行规定的... ..."

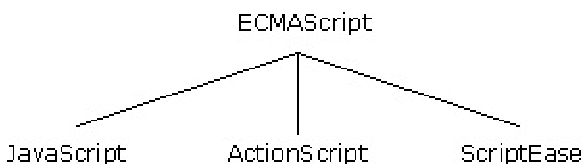
Web 浏览器对于 ECMAScript 来说是一个宿主环境，但它并不是唯一的宿主环境。事实上，还有不计其数的其他各种环境（例如 Nombas 的 ScriptEase，以及 Macromedia 同时用在 Flash 和 Director MX 中的 ActionScript）可以容纳 ECMAScript 实现。那么 ECMAScript 在浏览器之外规定了些什么呢？

简单地说，ECMAScript 描述了以下内容：

- 语法
- 类型

- 语句
- 关键字
- 保留字
- 运算符
- 对象

ECMAScript 仅仅是一个描述，定义了脚本语言的所有属性、方法和对象。其他语言可以实现 ECMAScript 来作为功能的基准，JavaScript 就是这样：



每个浏览器都有它自己的 ECMAScript 接口的实现，然后这个实现又被扩展，包含了 DOM 和 BOM（在以下几节中再探讨）。当然还有其他实现并扩展了 ECMAScript 的语言，例如 Windows 脚本宿主（Windows Scripting Host, WSH）、Macromedia 在 Flash 和 Director MX 中的 ActionScript，以及 Nombas ScriptEase。

1. ECMAScript 的版本

ECMAScript 分成几个不同的版本，它是在一个叫做 ECMA-262 的标准中定义的。和其他标准一样，ECMA-262 会被编辑和更新。当有了主要更新时，就会发布一个标准的新版。最新 ECMA-262 的版本是 5.1，于 2011 年 6 月发布。

ECMA-262 的第一版在本质上与 Netscape 的 JavaScript 1.1 是一样，只是把所有与浏览器相关的代码删除了，此外还有一些小的调整。首先，ECMA-262 要求对 Unicode 标准的支持（以便支持多语言）。第二，它要求对象是平台无关的（Netscape 的 JavaScript 1.1 事实上有不同的对象实现，例如 Date 对象，是依赖于平台）。这是 JavaScript 1.1 和 1.2 为什么不符合 ECMA-262 规范第一版的主要原因。

ECMA-262 的第二版大部分更新本质上是编辑性的。这次标准的更新是为了与 ISO/IEC-16262 的严格一致，也并没有特别添加、更改和删除内容。ECMAScript 一般不会遵守第二版。

ECMA-262 第三版是该标准第一次真正的更新。它提供了对字符串处理、错误定义和数值输出的更新。同时，它还增加了正则表达式、新的控制语句、try...catch 异常处理的支持，以及一些为使标准国际化而做的小改动。一般来说，它标志着 ECMAScript 成为一种真正的编程语言。

2. 何谓 ECMAScript 符合性

在 ECMA-262 中，ECMAScript 符合性（conformance）有明确的定义。一个脚本语言必须满足以下四项基本原则：

- 符合的实现必须按照 ECMA-262 中所描述的支持所有的“类型、值、对象、属性、函数和程序语言及语义”（ECMA-262，第一页）
- 符合的实现必须支持 Unicode 字符标准（UCS）
- 符合的实现可以增加没有在 ECMA-262 中指定的“额外类型、值、对象、属性和函数”。ECMA-262 将这些增加描述为规范中未给定的新对象或对象的新属性
- 符合的实现可以支持没有在 ECMA-262 中定义的“程序和正则表达式语法”（意思是可以替换或者扩展内建的正则表达式支持）

所有 ECMAScript 实现必须符合以上标准。

3. Web 浏览器中的 ECMAScript 支持

含有 JavaScript 1.1 的 Netscape Navigator 3.0 在 1996 年发布。然后，JavaScript 1.1 规范被作为一个新标准的草案被提交给 EMCA。有了 JavaScript 轰动性的流行，Netscape 十分高兴地开始开发 1.2 版。但有一个问题，ECMA 并未接受 Netscape 的草案。在 Netscape Navigator 3.0 发布后不久，微软就发布了 IE 3.0。该版本的 IE 含有 JScript 1.0（微软自己的 JavaScript 实现的名称），原本计划可以与 JavaScript 1.1 相提并论。然后，由于文档不全以及一些不当的重复特性，JScript 1.0 远远没有达到 JavaScript 1.1 的水平。

在 ECMA-262 第一版定稿之前，发布含有 JavaScript 1.2 的 Netscape Navigator 4.0 是在 1997 年，在那年晚些时候，ECMA-262 标准被接受并标准化。因此，JavaScript 1.2 并不和 ECMAScript 的第一版兼容，虽然 ECMAScript 应该基于 JavaScript 1.1。

JScript 的下一步是 IE 4.0 中加入的 JScript 3.0（2.0 版是随 IIS 3.0 一起发布的，但并未包含在浏览器中）。微软大力宣传 JScript 3.0 是世界上第一个真正符合 ECMA 标准的脚本语言。而那时，ECMA-262 还没有最终定稿，所以 JScript 3.0 也遭受了和 JavaScript 1.2 同样的命运 - 它还是没能符合最终的 ECMAScript 标准。

Netscape 选择在 Netscape Navigator 4.06 中升级它的 JavaScript 实现。JavaScript 1.3 使 Netscape 终于完全符合了 ECMAScript 第一版。Netscape 加入了对 Unicode 标准的支持，并让所有的对象保留了在 JavaScript 1.2 中引入的新特性的同时实现了平台独立。

当 Netscape 将它的源代码作为 Mozilla 项目公布于众时，本来计划 JavaScript 1.4 将会嵌入到 Netscape Navigator 5.0 中。然而，一个冒进的决定 - 要完全从头重新设计 Netscape 的代码，破坏了这个工作。JavaScript 1.4 仅仅作为一个 Netscape Enterprise Server 的服务器端脚本语言发布，以后也没有被放入浏览器中。

如今，所有主流的 Web 浏览器都遵守 ECMA-262 第三版。

下面的表格列出了大部分流行的 Web 浏览器中的 ECMAScript 支持：

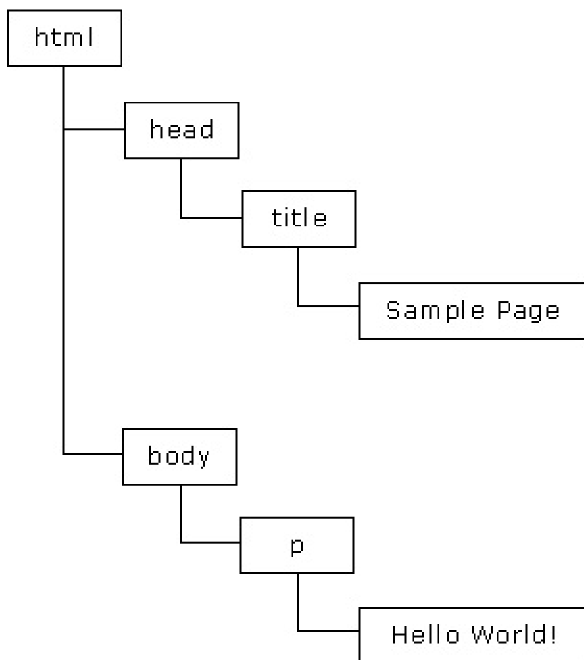
浏览器	DOM 兼容性
Netscape Navigator 2.0	-
Netscape Navigator 3.0	-
Netscape Navigator 4.0 - 4.05	-
Netscape Navigator 4.06 - 4.79	Edition 1
Netscape 6.0+ (Mozilla 0.6.0+)	Edition 3
Internet Explorer 3.0	-
Internet Explorer 4.0	-
Internet Explorer 5.0	Edition 1
Internet Explorer 5.5+	Edition 3
Opera 6.0 - 7.1	Edition 2
Opera 7.2+	Edition 3
Safari 1.0+/Konqueror ~ 2.0+	Edition 3

DOM

DOM（文档对象模型）是 HTML 和 XML 的应用程序接口（API）。DOM 将把整个页面规划成由节点层级构成的文档。HTML 或 XML 页面的每个部分都是一个节点的衍生物。请考虑下面的 HTML 页面：

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>hello world!</p>
  </body>
</html>
```

这段代码可以用 DOM 绘制成一个节点层次图：



DOM 通过创建树来表示文档，从而使开发者对文档的内容和结构具有空前的控制力。用 DOM API 可以轻松地删除、添加和替换节点。

1. 为什么 DOM 必不可少

自从 IE 4.0 和 Netscape Navigator 4.0 开始支持不同形态的动态 HTML（DHTML），开发者首次能够在不重载网页的情况下修改它的外观和内容。这是 Web 技术的一大飞跃，不过也带来了巨大的问题。Netscape 和微软各自开发自己的 DHTML，从而结束了 Web 开发者只编写一个 HTML 页面就可以在所有浏览器中访问的时期。

业界决定必须要做点什么以保持 Web 的跨平台特性，他们担心如果放任 Netscape 和微软公司这样做，Web 必将分化为两个独立的部分，每一部分只适用于特定的浏览器。因此，负责指定 Web 通信标准的团体 W3C（World Wide Web Consortium）就开始制定 DOM。

2. DOM 的各个 level

DOM Level 1 是 W3C 于 1998 年 10 月提出的。它由两个模块组成，即 DOM Core 和 DOM HTML。前者提供了基于 XML 的文档的结构图，以便访问和操作文档的任意部分；后者添加了一些 HTML 专用的对象和方法，从而扩展了 DOM Core。

注意，DOM 不是 JavaScript 专有的，事实上许多其他语言都实现了它。不过，Web 浏览器中的 DOM 已经用 ECMAScript 实现了，现在是 JavaScript 语言的一个很大组成部分。

DOM Level 1 只是一个目标，即规划文档的结构，DOM Level 2 的目标就广泛多了。对原始 DOM 的扩展添加了对鼠标和用户界面事件（DHTML 对此有丰富的支持）、范围、遍历（重复执行 DOM 文档的方法）的支持，并通过对象接口添加了对 CSS（层叠样式表）的支持。由 Level 1 引入的原始 DOM Core 也加入了对 XML 命名空间的支持。

DOM Level 2 引入了几种 DOM 新模块，用于处理新的接口类型：

- DOM 视图 - 描述跟踪文档的各种视图（即 CSS 样式化之前和 CSS 样式化之后的文档）
- DOM 事件 - 描述事件的接口
- DOM 样式 - 描述处理基于 CSS 样式的接口
- DOM 遍历和范围 - 描述遍历和操作文档树的接口

DOM Level 3 引入了以统一的方式载入和保持文档的方法（包含在新模块 DOM Load and Save）以及验证文档（DOM Validation）的方法，从而进一步扩展了 DOM。在 Level 3 中，DOM Core 被扩展为支持所有的 XML 1.0 特性，包括 XML Infoset、XPath 和 XML Base。

在学习 DOM 时，可能会遇到有人引用 DOM Level 0。注意，根本没有 DOM Level 0 这个标准，它只是 DOM 的一个历史参考点（DOM Level 0 指的是 IE 4.0 和 Netscape Navigator 4.0 中支持的原始 DHTML）。

3. 其他 DOM

除了 DOM Core 和 DOM HTML 外，还有其他几种语言发布了自己的 DOM 标准。这些语言都是基于 XML 的，每种 DOM 都给对应语言添加了特有的方法和接口：

- 可缩放矢量语言（SVG）1.0
- 数字标记语言（MathML）1.0
- 同步多媒体集成语言（SMIL）

注释：如果希望学习更多相关内容，请访问 w3school 的 SMIL 教程 和 SVG 教程。

此外，其他语言也开发了自己的 DOM 实现，如 Mozilla 的 XML 用户界面语言（XUL）。不过，只有上面列出的几种语言是 W3C 的推荐标准。

4. Web 浏览器中的 DOM 支持

DOM 在被 Web 浏览器开始实现之前就已经是一种标准了。IE 首次尝试 DOM 是在 5.0 版本中，不过其实直到 5.5 版本之后才具有真正的 DOM 支持，IE 5.5 实现了 DOM Level 1。从那时起，IE 就没有引入新的 DOM 功能。

Netscape 直到 Netscape 6（Mozilla 0.6.0）才引入 DOM 支持。目前，Mozilla 具有最好的 DOM 支持，实现了完整的 Level 1、几乎所有 Level 2 以及一部分 Level 3。（Mozilla 开发小组的目标是构造一个与标准 100% 兼容的浏览器，他们的工作得到了回报。）

Opera 直到 7.0 版本才加入 DOM 支持，还有 Safari 也实现了大部分 DOM Level 1。它们几乎都与 IE 5.5 处于同一水平，有些情况下，甚至超过了 IE 5.5。不过，就对 DOM 的支持而论，所有浏览器都远远落后于 Mozilla。下表列出了常用浏览器对 DOM 的支持。

浏览器	DOM 兼容性
Netscape Navigator 1.0 - 4.x	-
Netscape 6.0+ (Mozilla 0.6.0+)	Level 1、Level 2、Level 3（部分）
IE 2.0 - 4.x	-
IE 5.0	Level 1（最小）
IE 5.5+	Level 1（几乎全部）
Opera 1.0 - 6.0	-
Opera 7.0+	Level 1（几乎全部）、Level 2 （部分）
Safari 1.0+/Konqueror ~ 2.0+	Level 1

注释：如果希望进一步地学习 DOM 的知识，请访问 w3school 的 HTML DOM 教程 和 XML DOM 教程。

BOM

IE 3.0 和 Netscape Navigator 3.0 提供了一种特性 - BOM（浏览器对象模型），可以对浏览器窗口进行访问和操作。使用 BOM，开发者可以移动窗口、改变状态栏中的文本以及执行其他与页面内容不直接相关的动作。使 BOM 独树一帜且又常常令人怀疑的地方在于，它只是 JavaScript 的一个部分，没有任何相关的标准。

BOM 主要处理浏览器窗口和框架，不过通常浏览器特定的 JavaScript 扩展都被看做 BOM 的一部分。这些扩展包括：

- 弹出新的浏览器窗口
- 移动、关闭浏览器窗口以及调整窗口大小
- 提供 Web 浏览器详细信息的定位对象
- 提供用户屏幕分辨率详细信息的屏幕对象
- 对 cookie 的支持
- IE 扩展了 BOM，加入了 ActiveXObject 类，可以通过 JavaScript 实例化 ActiveX 对象

由于没有相关的 BOM 标准，每种浏览器都有自己的 BOM 实现。有一些事实上的标准，如具有一个窗口对象和一个导航对象，不过每种浏览器可以为这些对象或其他对象定义自己的属性和方法。

参阅：

- Window 对象
- Navigator 对象
- Screen 对象

- History 对象
- Location 对象

ECMAScript 基础

ECMAScript 语法

熟悉 **Java**、**C** 和 **Perl** 这些语言的开发者会发现 **ECMAScript** 的语法很容易掌握，因为它借用了这些语言的语法。

Java 和 **ECMAScript** 有一些关键的语法特性相同，也有一些完全不同。

区分大小写

与 Java 一样，变量、函数名、运算符以及其他一切东西都是区分大小写的。

比如：

变量 test 与变量 TEST 是不同的。

变量是弱类型的

与 Java 和 C 不同，ECMAScript 中的变量无特定的类型，定义变量时只用 var 运算符，可以将它初始化为任意值。

因此，可以随时改变变量所存数据的类型（尽量避免这样做）。

例子

```
var color = "red";
var num = 25;
var visible = true;
```

每行结尾的分号可有可无

Java、C 和 Perl 都要求每行代码以分号（;）结束才符合语法。

ECMAScript 则允许开发者自行决定是否以分号结束一行代码。如果没有分号，ECMAScript 就把折行代码的结尾看做该语句的结尾（与 Visual Basic 和 VBScript 相似），前提是这样没有破坏代码的语义。

最好的代码编写习惯是总加入分号，因为没有分号，有些浏览器就不能正确运行，不过根据 ECMAScript 标准，下面两行代码都是正确的：

```
var test1 = "red"
var test2 = "blue";
```

注释与 Java、C 和 PHP 语言的注释相同

ECMAScript 借用了这些语言的注释语法。

有两种类型的注释：

- 单行注释以双斜杠开头（//）
- 多行注释以单斜杠和星号开头（/），以星号和单斜杠结尾（/）

```
//this is a single-line comment

/*this is a multi-
line comment*/
```

括号表示代码块

从 Java 中借鉴的另一个概念是代码块。

代码块表示一系列应该按顺序执行的语句，这些语句被封装在左括号（{）和右括号（}）之间。

例如：

```
if (test1 == "red") {
    test1 = "blue";
    alert(test1);
}
```

ECMAScript 变量

请使用 **var** 运算符声明变量。

变量名需要遵守一些简单的规则。

声明变量

在上一节中我们讲解过，ECMAScript 中的变量是用 var 运算符（variable 的缩写）加变量名定义的。例如：

```
var test = "hi";
```

在这个例子中，声明了变量 test，并把它的值初始化为 "hi"（字符串）。由于 ECMAScript 是弱类型的，所以解释程序会为 test 自动创建一个字符串值，无需明确的类型声明。

还可以用一个 var 语句定义两个或多个变量：

```
var test1 = "hi", test2 = "hello";
```

前面的代码定义了变量 test1，初始值为 "hi"，还定义了变量 test2，初始值为 "hello"。

不过用同一个 *var* 语句定义的变量不必具有相同的类型，如下所示：

```
var test = "hi", age = 25;
```

这个例子除了（再次）定义 test 外，还定义了 age，并把它初始化为 25。即使 test 和 age 属于两种不同的数据类型，在 ECMAScript 中这样定义也是完全合法的。

与 Java 不同，ECMAScript 中的变量并不一定要初始化（它们是在幕后初始化的，将在后面讨论这一点）。因此，下面这一行代码也是有效的：

```
var test;
```

此外，与 Java 不同的还有变量可以存放不同类型的值。这是弱类型变量的优势。例如，可以把变量初始化为字符串类型的值，之后把它设置为数字值，如下所示：

```
var test = "hi";
alert(test);
test = 55;
alert(test);
```

这段代码将毫无问题地输出字符串值和数字值。但是，如前所述，使用变量时，好的编码习惯是始终存放相同类型的值。

命名变量

变量名需要遵守两条简单的规则：

- 第一个字符必须是字母、下划线（_）或美元符号（$）
- 余下的字符可以是下划线、美元符号或任何字母或数字字符

下面的变量都是合法的：

```
var test;
var $test;
var $1;
var _$te$t2;
```

著名的变量命名规则

只是因为变量名的语法正确，并不意味着就该使用它们。变量还应遵守以下某条著名的命名规则：

Camel 标记法

首字母是小写的，接下来的字母都以大写字符开头。例如：

```
var myTestValue = 0, mySecondValue = "hi";
```

Pascal 标记法

首字母是大写的，接下来的字母都以大写字符开头。例如：

```
var MyTestValue = 0, MySecondValue = "hi";
```

匈牙利类型标记法

在以 Pascal 标记法命名的变量前附加一个小写字母（或小写字母序列），说明该变量的类型。例如，i 表示整数，s 表示字符串，如下所示“

```
var iMyTestValue = 0, sMySecondValue = "hi";
```

本教程采用了这些前缀，以使示例代码更易阅读：

类型	前缀	示例
数组	a	aValues
布尔型	b	bFound
浮点型（数字）	f	fValue
函数	fn	fnMethod
整型（数字）	i	iValue
对象	o	oType
正则表达式	re	rePattern
字符串	s	sValue
变型（可以是任何类型）	v	vValue

变量声明不是必须的

ECMAScript 另一个有趣的方面（也是与大多数程序设计语言的主要区别），是在使用变量之前不必声明。例如：

```
var sTest = "hello ";
sTest2 = sTest + "world";
alert(sTest2);
```

在上面的代码中，首先，sTest 被声明为字符串类型的值 "hello"。接下来的一行，用变量 sTest2 把 sTest 与字符串 "world" 连在一起。变量 sTest2 并没有用 var 运算符定义，这里只是插入了它，就像已经声明过它一样。

ECMAScript 的解释程序遇到未声明过的标识符时，用该变量名创建一个全局变量，并将其初始化为指定的值。

这是该语言的便利之处，不过如果不能紧密跟踪变量，这样做也很危险。最好的习惯是像使用其他程序设计语言一样，总是声明所有变量。

ECMAScript 关键字

本节提供完整的 **ECMAScript** 关键字列表。

ECMAScript 关键字

ECMA-262 定义了 ECMAScript 支持的一套关键字（*keyword*）。

这些关键字标识了 ECMAScript 语句的开头和/或结尾。根据规定，关键字是保留的，不能用作变量名或函数名。

下面是 ECMAScript 关键字的完整列表：

```
break
case
catch
continue
default
delete
do
else
finally
for
function
if
in
instanceof
new
return
switch
this
throw
try
typeof
var
void
while
with
```

注意：如果把关键字用作变量名或函数名，可能得到诸如 "Identifier Expected"（应该有标识符、期望标识符）这样的错误消息。

ECMAScript 保留字

本节提供完整的 **ECMAScript** 保留字列表。

ECMAScript 保留字

ECMA-262 定义了 ECMAScript 支持的一套保留字（*reserved word*）。

保留字在某种意思上是为将来的关键字而保留的单词。因此保留字不能被用作变量名或函数名。

ECMA-262 第三版中保留字的完整列表如下：

```
abstract
boolean
byte
char
class
const
debugger
double
enum
export
extends
final
float
goto
implements
import
int
interface
long
native
package
private
protected
public
short
static
super
synchronized
throws
transient
volatile
```

注意：如果将保留字用作变量名或函数名，那么除非将来的浏览器实现了该保留字，否则很可能收不到任何错误消息。当浏览器将其实现后，该单词将被看做关键字，如此将出现关键字错误。

ECMAScript 原始值和引用值

在 **ECMAScript** 中，变量可以存在两种类型的值，即原始值和引用值。

原始值和引用值

在 ECMAScript 中，变量可以存在两种类型的值，即原始值和引用值。

原始值

存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

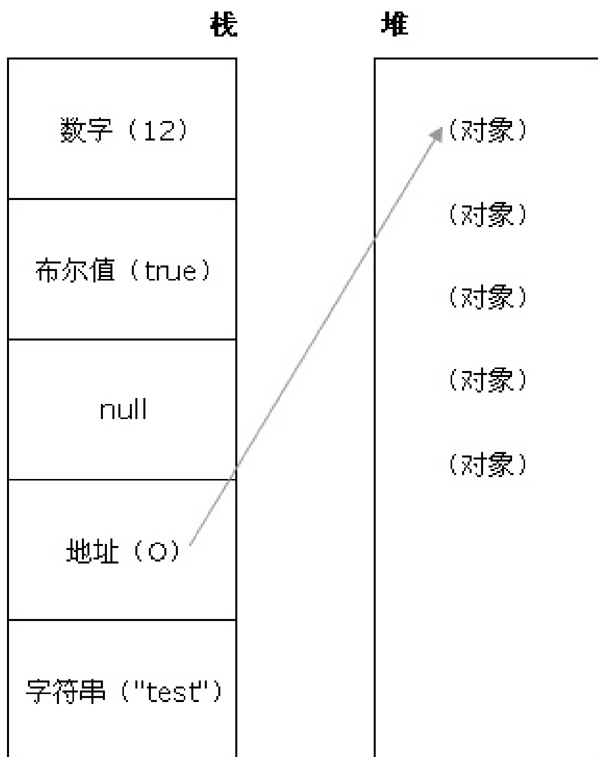
引用值

存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存处。

为变量赋值时，ECMAScript 的解释程序必须判断该值是原始类型，还是引用类型。要实现这一点，解释程序则需尝试判断该值是否为 *ECMAScript* 的原始类型之一，即 Undefined、Null、Boolean、Number 和 String 型。由于这些原始类型占据的空间是固定的，所以可将他们存储在较小的内存区域 - 栈中。这样存储便于迅速查寻变量的值。

在许多语言中，字符串都被看作引用类型，而非原始类型，因为字符串的长度是可变的。**ECMAScript** 打破了这一传统。

如果一个值是引用类型的，那么它的存储空间将从堆中分配。由于引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。如下图所示：



原始类型

如前所述，ECMAScript 有 5 种原始类型（primitive type），即 Undefined、Null、Boolean、Number 和 String。ECMA-262 把术语类型（*type*）定义为值的一个集合，每种原始类型定义了它包含的值的范围及其字面量表示形式。

ECMAScript 提供了 typeof 运算符来判断一个值是否在某种类型的范围内。可以用这种运算符判断一个值是否表示一种原始类型：如果它是原始类型，还可以判断它表示哪种原始类型。

在稍后的章节，我们将为您深入讲解 ECMAScript 的原始类型和引用类型。

ECMAScript 原始类型

ECMAScript 有 5 种原始类型（primitive type），即 Undefined、Null、Boolean、Number 和 String。

typeof 运算符

typeof 运算符有一个参数，即要检查的变量或值。例如：

```
var sTemp = "test string";
alert (typeof sTemp);    //输出 "string"
alert (typeof 86);    //输出 "number"
```

对变量或值调用 typeof 运算符将返回下列值之一：

- undefined - 如果变量是 Undefined 类型的
- boolean - 如果变量是 Boolean 类型的
- number - 如果变量是 Number 类型的
- string - 如果变量是 String 类型的
- object - 如果变量是一种引用类型或 Null 类型的

注释：您也许会问，为什么 typeof 运算符对于 null 值会返回 "Object"。这实际上是 JavaScript 最初实现中的一个错误，然后被 ECMAScript 沿用了。现在，null 被认为是对象的占位符，从而解释了这一矛盾，但从技术上来说，它仍然是原始值。

Undefined 类型

如前所述，Undefined 类型只有一个值，即 undefined。当声明的变量未初始化时，该变量的默认值是 undefined。

```
var oTemp;
```

前面一行代码声明变量 oTemp，没有初始值。该变量将被赋予值 undefined，即 undefined 类型的字面量。可以用下面的代码段测试该变量的值是否等于 undefined：

```
var oTemp;
alert(oTemp == undefined);
```

这段代码将显示 "true"，说明这两个值确实相等。还可以用 typeof 运算符显示该变量的值是 undefined：

```
var oTemp;
alert(typeof oTemp); //输出 "undefined"
```

提示：值 undefined 并不同于未定义的值。但是，typeof 运算符并不真正区分这两种值。考虑下面的代码：

```
var oTemp;

alert(typeof oTemp);  //输出 "undefined"
alert(typeof oTemp2);  //输出 "undefined"
```

前面的代码对两个变量输出的都是 "undefined"，即使只有变量 oTemp2 从未被声明过。如果对 oTemp2 使用除 typeof 之外的其他运算符的话，会引起错误，因为其他运算符只能用于已声明的变量上。

例如，下面的代码将引发错误：

```
var oTemp;
alert(oTemp2 == undefined);
```

当函数无明确返回值时，返回的也是值 "undefined"，如下所示：

```
function testFunc() {
}

alert(testFunc() == undefined);  //输出 "true"
```

Null 类型

另一种只有一个值的类型是 Null，它只有一个专用值 null，即它的字面量。值 undefined 实际上是从值 null 派生来的，因此 ECMAScript 把它们定义为相等的。

```
alert(null == undefined);  //输出 "true"
```

尽管这两个值相等，但它们的含义不同。undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象（在讨论 typeof 运算符时，简单地介绍过这一点）。如果函数或方法要返回的是对象，那么找不到该对象时，返回的通常是 null。

Boolean 类型

Boolean 类型是 ECMAScript 中最常用的类型之一。它有两个值 true 和 false （即两个 Boolean 字面量）。

即使 false 不等于 0，0 也可以在必要时被转换成 false，这样在 Boolean 语句中使用两者都是安全的。

```
var bFound = true;
var bLost = false;
```

Number 类型

ECMA-262 中定义的最特殊的类型是 Number 类型。这种类型既可以表示 32 位的整数，还可以表示 64 位的浮点数。

直接输入的（而不是从另一个变量访问的）任何数字都被看做 Number 类型的字面量。例如，下面的代码声明了存放整数值的变量，它的值由字面量 86 定义：

```
var iNum = 86;
```

八进制数和十六进制数

整数也可以被表示为八进制（以 8 为底）或十六进制（以 16 为底）的字面量。八进制字面量的首数字必须是 0，其后的数字可以是任何八进制数字（0-7），如下面的代码所示：

```
var iNum = 070;  //070 等于十进制的 56
```

要创建十六进制的字面量，首位数字必须为 0，后面接字母 x，然后是任意的十六进制数字（0 到 9 和 A 到 F）。这些字母可以是大写的，也可以是小写的。例如：

```
var iNum = 0x1f;  //0x1f 等于十进制的 31
var iNum = 0xAB;  //0xAB 等于十进制的 171
```

提示：尽管所有整数都可以表示为八进制或十六进制的字面量，但所有数学运算返回的都是十进制结果。

浮点数

要定义浮点值，必须包括小数点和小数点后的一位数字（例如，用 1.0 而不是 1）。这被看作浮点数字面量。例如：

```
var fNum = 5.0;
```

对于浮点字面量的有趣之处在于，用它进行计算前，真正存储的是字符串。

科学计数法

对于非常大或非常小的数，可以用科学计数法表示浮点数，可以把一个数表示为数字（包括十进制数字）加 e（或 E），后面加乘以 10 的倍数。例如：

```
var fNum = 5.618e7
```

该符号表示的是数 56180000。把科学计数法转化成计算式就可以得到该值：5.618 x 10⁷。

也可以用科学计数法表示非常小的数，例如 0.00000000000000008 可以表示为 8-e¹⁷（这里，10 被升到 -17 次冥，意味着需要被 10 除 17 次）。ECMAScript 默认把具有 6 个或 6 个以上前导 0 的浮点数转换成科学计数法。

提示：也可用 64 位 IEEE 754 形式存储浮点值，这意味着十进制值最多可以有 17 个十进制位。17 位之后的值将被裁去，从而造成一些小的数学误差。

特殊的 **Number** 值

几个特殊值也被定义为 Number 类型。前两个是 Number.MAX_VALUE 和 Number.MIN_VALUE，它们定义了 Number 值集合的外边界。所有 ECMAScript 数都必须在这两个值之间。不过计算生成的数值结果可以不落在这两个值之间。

当计算生成的数大于 Number.MAX_VALUE 时，它将被赋予值 Number.POSITIVE_INFINITY，意味着不再有数字值。同样，生成的数值小于 Number.MIN_VALUE 的计算也会被赋予值 Number.NEGATIVE_INFINITY，也意味着不再有数字值。如果计算返回的是无穷大值，那么生成的结果不能再用于其他计算。

事实上，有专门的值表示无穷大，（如你猜到的）即 Infinity。Number.POSITIVE_INFINITY 的值为 Infinity。Number.NEGATIVE_INFINITY 的值为 -Infinity。

由于无穷大数可以是正数也可以是负数，所以可用一个方法判断一个数是否是有穷的（而不是单独测试每个无穷数）。可以对任何数调用 isFinite() 方法，以确保该数不是无穷大。例如：

```
var iResult = iNum * some_really_large_number;

`if (isFinite(iResult))` {
    alert("finite");
}

else {
    alert("infinite");
}
```

最后一个特殊值是 NaN，表示非数（Not a Number）。NaN 是个奇怪的特殊值。一般说来，这种情况发生在类型（String、Boolean 等）转换失败时。例如，要把单词 blue 转换成数值就会失败，因为没有与之等价的数值。与无穷大一样，NaN 也不能用于算术计算。NaN 的另一个奇特之处在于，它与自身不相等，这意味着下面的代码将返回 false：

```
alert(NaN == NaN);  //输出 "false"
```

出于这个原因，不推荐使用 NaN 值本身。函数 isNaN() 会做得相当好：

```
alert(isNaN("blue"));  //输出 "true"
alert(isNaN("666"));  //输出 "false"
```

String 类型

String 类型的独特之处在于，它是唯一没有固定大小的原始类型。可以用字符串存储 0 或更多的 Unicode 字符，有 16 位整数表示（Unicode 是一种国际字符集，本教程后面将讨论它）。

字符串中每个字符都有特定的位置，首字符从位置 0 开始，第二个字符在位置 1，依此类推。这意味着字符串中的最后一个字符的位置一定是字符串的长度减 1：

这个字符串的长度是 6

位置	H	E	L	L	O	!
	0	1	2	3	4	5

字符串字面量是由双引号（"）或单引号（'）声明的。而 Java 则是用双引号声明字符串，用单引号声明字符。但是由于 ECMAScript 没有字符类型，所以可使用这两种表示法中的任何一种。例如，下面的两行代码都有效：

```
var sColor1 = "red";
var sColor2 = 'red';
```

String 类型还包括几种字符字面量，Java、C 和 Perl 的开发者应该对此非常熟悉。

下面列出了 ECMAScript 的字符字面量：

字面量	含义
\n	换行
\t	制表符
\b	空格
\r	回车
\f	换页符
\	反斜杠
\'	单引号
\"	双引号
\0*nnn*	八进制代码 *nnn* 表示的字符（*n* 是 0 到 7 中的一个八进制数字）
\x*nn*	十六进制代码 *nn* 表示的字符（*n* 是 0 到 F 中的一个十六进制数字）
\u*nnnn*	十六进制代码 *nnnn* 表示的 Unicode 字符（*n* 是 0 到 F 中的一个十六进制数字）

ECMAScript 类型转换

所有程序设计语言最重要的特征之一是具有进行类型转换的能力。

ECMAScript 给开发者提供了大量简单的类型转换方法。

大部分类型具有进行简单转换的方法，还有几个全局方法可以用于更复杂的转换。无论哪种情况，在 **ECMAScript** 中，类型转换都是简短的一步操作。

转换成字符串

ECMAScript 的 Boolean 值、数字和字符串的原始值的有趣之处在于它们是伪对象，这意味着它们实际上具有属性和方法。

例如，要获得字符串的长度，可以采用下面的代码：

```
var sColor = "red";
alert(sColor.length);    //输出 "3"
```

尽管 "red" 是原始类型的字符串，它仍然具有属性 length，用于存放字符串的大小。

总而言之，3 种主要的原始类型 Boolean 值、数字和字符串都有 toString() 方法，可以把它们的值转换成字符串。

提示：您也许会问，“字符串还有 toString() 方法吗，这不是多余吗？”是的，的确如此，不过 ECMAScript 定义所有对象都有 toString() 方法，无论它是伪对象，还是真对象。因为 String 类型属于伪对象，所以它一定有 toString() 方法。

Boolean 类型的 toString() 方法只是输出 "true" 或 "false"，结果由变量的值决定：

```
var bFound = false;
alert(bFound.toString());    //输出 "false"
```

Number 类型的 toString() 方法比较特殊，它有两种模式，即默认模式和基模式。采用默认模式，toString() 方法只是用相应的字符串输出数字值（无论是整数、浮点数还是科学计数法），如下所示：

```
var iNum1 = 10;
var iNum2 = 10.0;
alert(iNum1.toString());    //输出 "10"
alert(iNum2.toString());    //输出 "10"
```

注释：在默认模式中，无论最初采用什么表示法声明数字，Number 类型的 toString() 方法返回的都是数字的十进制表示。因此，以八进制或十六进制字面量形式声明的数字输出的都是十进制形式的。

采用 Number 类型的 toString() 方法的基模式，可以用不同的基输出数字，例如二进制的基是 2，八进制的基是 8，十六进制的基是 16。

基只是要转换成的基数的另一种加法而已，它是 toString() 方法的参数：

```
var iNum = 10;
alert(iNum.toString(2));    //输出 "1010"
alert(iNum.toString(8));    //输出 "12"
alert(iNum.toString(16));    //输出 "A"
```

在前面的示例中，以 3 种不同的形式输出了数字 10，即二进制形式、八进制形式和十六进制形式。HTML 采用十六进制表示每种颜色，在 HTML 中处理数字时这种功能非常有用。

注释：对数字调用 toString(10) 与调用 toString() 相同，它们返回的都是该数字的十进制形式。

参阅：

请参阅 JavaScript 参考手册提供的有关 toString() 方法的详细信息：

- arrayObject.toString()
- booleanObject.toString()
- dateObject.toString()
- NumberObject.toString()
- stringObject.toString()

转换成数字

ECMAScript 提供了两种把非数字的原始值转换成数字的方法，即 parseInt() 和 parseFloat()。

正如您可能想到的，前者把值转换成整数，后者把值转换成浮点数。只有对 String 类型调用这些方法，它们才能正确运行；对其他类型返回的都是 NaN。

parseInt()

在判断字符串是否是数字值前，parseInt() 和 parseFloat() 都会仔细分析该字符串。

parseInt() 方法首先查看位置 0 处的字符，判断它是否是个有效数字；如果不是，该方法将返回 NaN，不再继续执行其他操作。但如果该字符是有效数字，该方法将查看位置 1 处的字符，进行同样的测试。这一过程将持续到发现非有效数字的字符为止，此时 parseInt() 将把该

字符之前的字符串转换成数字。

例如，如果要把字符串 "12345red" 转换成整数，那么 parseInt() 将返回 12345，因为当它检查到字符 r 时，就会停止检测过程。

字符串中包含的数字字面量会被正确转换为数字，比如 "0xA" 会被正确转换为数字 10。不过，字符串 "22.5" 将被转换成 22，因为对于整数来说，小数点是无效字符。

一些示例如下：

```
var iNum1 = parseInt("12345red");	//返回 12345
var iNum1 = parseInt("0xA");	//返回 10
var iNum1 = parseInt("56.9");	//返回 56
var iNum1 = parseInt("red");	//返回 NaN
```

parseInt() 方法还有基模式，可以把二进制、八进制、十六进制或其他任何进制的字符串转换成整数。基是由 parseInt() 方法的第二个参数指定的，所以要解析十六进制的值，需如下调用 parseInt() 方法：

```
var iNum1 = parseInt("AF", 16);	//返回 175
```

当然，对二进制、八进制甚至十进制（默认模式），都可以这样调用 parseInt() 方法：

```
var iNum1 = parseInt("10", 2);	//返回 2
var iNum2 = parseInt("10", 8);	//返回 8
var iNum3 = parseInt("10", 10);	//返回 10
```

如果十进制数包含前导 0，那么最好采用基数 10，这样才不会意外地得到八进制的值。例如：

```
var iNum1 = parseInt("010");	//返回 8
var iNum2 = parseInt("010", 8);	//返回 8
var iNum3 = parseInt("010", 10);	//返回 10
```

在这段代码中，两行代码都把字符 "010" 解析成一个数字。第一行代码把这个字符串看作八进制的值，解析它的方式与第二行代码（声明基数为 8）相同。最后一行代码声明基数为 10，所以 iNum3 最后等于 10。

参阅

请参阅 JavaScript 参考手册提供的有关 parseInt() 方法的详细信息：parseInt()。

parseFloat()

parseFloat() 方法与 parseInt() 方法的处理方式相似，从位置 0 开始查看每个字符，直到找到第一个非有效的字符为止，然后把该字符之前的字符串转换成整数。

不过，对于这个方法来说，第一个出现的小数点是有效字符。如果有两个小数点，第二个小数点将被看作无效的。parseFloat() 会把这个小数点之前的字符转换成数字。这意味着字符串 "11.22.33" 将被解析成 11.22。

使用 parseFloat() 方法的另一不同之处在于，字符串必须以十进制形式表示浮点数，而不是用八进制或十六进制。该方法会忽略前导 0，所以八进制数 0102 将被解析为 102。对于十六进制数 0xA，该方法将返回 NaN，因为在浮点数中，x 不是有效字符。（注释：经测试，具体的浏览器实现会返回 0，而不是 NaN。）

此外，parseFloat() 方法也没有基模式。

下面是使用 parseFloat() 方法的一些示例：

```
var fNum1 = parseFloat("12345red");    //返回 12345
var fNum2 = parseFloat("0xA");    //返回 NaN
var fNum3 = parseFloat("11.2");    //返回 11.2
var fNum4 = parseFloat("11.22.33");    //返回 11.22
var fNum5 = parseFloat("0102");    //返回 102
var fNum1 = parseFloat("red");    //返回 NaN
```

参阅

请参阅 JavaScript 参考手册提供的有关 parseFloat() 方法的详细信息：parseFloat()。

强制类型转换

您还可以使用强制类型转换（*type casting*）来处理转换值的类型。使用强制类型转换可以访问特定的值，即使它是另一种类型的。

编者注：cast 有“铸造”之意，很贴合“强制转换”的意思。

ECMAScript 中可用的 3 种强制类型转换如下：

- Boolean(value) - 把给定的值转换成 Boolean 型；
- Number(value) - 把给定的值转换成数字（可以是整数或浮点数）；
- String(value) - 把给定的值转换成字符串；

用这三个函数之一转换值，将创建一个新值，存放由原始值直接转换成的值。这会造成意想不到的后果。

Boolean() 函数

当要转换的值是至少有一个字符的字符串、非 0 数字或对象时，Boolean() 函数将返回 true。如果该值是空字符串、数字 0、undefined 或 null，它将返回 false。

可以用下面的代码测试 Boolean 型的强制类型转换：

```
var b1 = Boolean("");          //false - 空字符串
var b2 = Boolean("hello");          //true - 非空字符串
var b1 = Boolean(50);          //true - 非零数字
var b1 = Boolean(null);          //false - null
var b1 = Boolean(0);          //false - 零
var b1 = Boolean(new object());     //true - 对象
```

Number() 函数

Number() 函数的强制类型转换与 parseInt() 和 parseFloat() 方法的处理方式相似，只是它转换的是整个值，而不是部分值。

还记得吗，parseInt() 和 parseFloat() 方法只转换第一个无效字符之前的字符串，因此 "1.2.3" 将分别被转换为 "1" 和 "1.2"。

用 Number() 进行强制类型转换，"1.2.3" 将返回 NaN，因为整个字符串值不能转换成数字。如果字符串值能被完整地转换，Number() 将判断是调用 parseInt() 方法还是 parseFloat() 方法。

下表说明了对不同的值调用 Number() 方法会发生的情况：

用法	结果
Number(false)	0
Number(true)	1
Number(undefined)	NaN
Number(null)	0
Number("1.2")	1.2
Number("12")	12
Number("1.2.3")	NaN
Number(new object())	NaN
Number(50)	50

String() 函数

最后一种强制类型转换方法 String() 是最简单的，因为它可把任何值转换成字符串。

要执行这种强制类型转换，只需要调用作为参数传递进来的值的 toString() 方法，即把 12 转换成 "12"，把 true 转换成 "true"，把 false 转换成 "false"，以此类推。

强制转换成字符串和调用 toString() 方法的唯一不同之处在于，对 null 和 undefined 值强制类型转换可以生成字符串而不引发错误：

```
var s1 = String(null);    //"null"
var oNull = null;
var s2 = oNull.toString();    //会引发错误
```

在处理 ECMAScript 这样的弱类型语言时，强制类型转换非常有用，不过应该确保使用值的正确。

ECMAScript 引用类型

引用类型通常叫做类（**class**）。

本教程会讨论大量的 **ECMAScript** 预定义引用类型。

引用类型

引用类型通常叫做类（class），也就是说，遇到引用值，所处理的就是对象。

本教程会讨论大量的 ECMAScript 预定义引用类型。

从现在起，将重点讨论与已经讨论过的原始类型紧密相关的引用类型。

注意：从传统意义上来说，ECMAScript 并不真正具有类。事实上，除了说明不存在类，在 ECMA-262 中根本没有出现“类”这个词。ECMAScript 定义了“对象定义”，逻辑上等价于其他程序设计语言中的类。

提示：本教程将使用术语“对象”。

对象是由 new 运算符加上要实例化的对象的名字创建的。例如，下面的代码创建 Object 对象的实例：

```
var o = new Object();
```

这种语法与 Java 语言的相似，不过当有不止一个参数时，ECMAScript 要求使用括号。如果没有参数，如以下代码所示，括号可以省略：

```
var o = new Object;
```

注意：尽管括号不是必需的，但是为了避免混乱，最好使用括号。

提示：我们会在对象基础这一章中更深入地探讨对象及其行为。

这一节的重点是具有等价的原始类型的引用类型。

Object 对象

Object 对象自身用处不大，不过在了解其他类之前，还是应该了解它。因为 ECMAScript 中的 Object 对象与 Java 中的 java.lang.Object 相似，ECMAScript 中的所有对象都由这个对象继承而来，Object 对象中的所有属性和方法都会出现在其他对象中，所以理解了 Object 对象，就可以更好地理解其他对象。

Object 对象具有下列属性：

constructor

对创建对象的函数的引用（指针）。对于 Object 对象，该指针指向原始的 Object() 函数。

Prototype

对该对象的对象原型的引用。对于所有的对象，它默认返回 Object 对象的一个实例。

Object 对象还具有几个方法：

hasOwnProperty(property)

判断对象是否有某个特定的属性。必须用字符串指定该属性。（例如，o.hasOwnProperty("name")）

IsPrototypeOf(object)

判断该对象是否为另一个对象的原型。

PropertyIsEnumerable

判断给定的属性是否可以用 for...in 语句进行枚举。

ToString()

返回对象的原始字符串表示。对于 Object 对象，ECMA-262 没有定义这个值，所以不同的 ECMAScript 实现具有不同的值。

ValueOf()

返回最适合该对象的原始值。对于许多对象，该方法返回的值都与 ToString() 的返回值相同。

注释：上面列出的每种属性和方法都会被其他对象覆盖。

Boolean 对象

Boolean 对象是 Boolean 原始类型的引用类型。

要创建 Boolean 对象，只需要传递 Boolean 值作为参数：

```
var oBooleanObject = new Boolean(true);
```

Boolean 对象将覆盖 Object 对象的 ValueOf() 方法，返回原始值，即 true 和 false。ToString() 方法也会被覆盖，返回字符串 "true" 或 "false"。

遗憾的是，在 ECMAScript 中很少使用 Boolean 对象，即使使用，也不易理解。

问题通常出现在 Boolean 表达式中使用 Boolean 对象时。例如：

```
var oFalseObject = new Boolean(false);
var bResult = oFalseObject && true;    //输出 true
```

在这段代码中，用 false 值创建 Boolean 对象。然后用这个值与原始值 true 进行 AND 操作。在 Boolean 运算中，false 和 true 进行 AND 操作的结果是 false。不过，在这行代码中，计算的是 oFalseObject，而不是它的值 false。

正如前面讨论过的，在 Boolean 表达式中，所有对象都会被自动转换为 true，所以 oFalseObject 的值是 true。然后 true 再与 true 进行 AND 操作，结果为 true。

注意：虽然你应该了解 Boolean 对象的可用性，不过最好还是使用 Boolean 原始值，避免发生这一节提到的问题。

参阅

如需更多有关 Boolean 对象的信息，请访问 JavaScript Boolean 对象参考手册。

Number 对象

正如你可能想到的，Number 对象是 Number 原始类型的引用类型。要创建 Number 对象，采用下列代码：

```
var oNumberObject = new Number(68);
```

您应该已认出本章前面小节中讨论特殊值（如 Number.MAX_VALUE）时提到的 Number 对象。所有特殊值都是 Number 对象的静态属性。

要得到数字对象的 Number 原始值，只需要使用 valueOf() 方法：

```
var iNumber = oNumberObject.valueOf();
```

当然，Number 类也有 toString() 方法，在讨论类型转换的小节中已经详细讨论过该方法。

除了从 Object 对象继承的标准方法外，Number 对象还有几个处理数值的专用方法。

toFixed() 方法

toFixed() 方法返回的是具有指定位数小数的数字的字符串表示。例如：

```
var oNumberObject = new Number(68);
alert(oNumberObject.toFixed(2));  //输出 "68.00"
```

在这里，toFixed() 方法的参数是 2，说明应该显示两位小数。该方法返回 "68.00"，空的字符串位由 0 来补充。对于处理货币的应用程序，该方法非常有用。toFixed() 方法能表示具有 0 到 20 位小数的数字，超过这个范围的值会引发错误。

toExponential() 方法

与格式化数字相关的另一个方法是 toExponential()，它返回的是用科学计数法表示的数字的字符串形式。

与 toFixed() 方法相似，toExponential() 方法也有一个参数，指定要输出的小数的位数。例如：

```
var oNumberObject = new Number(68);
alert(oNumberObject.toExponential(1));  //输出 "6.8e+1"
```

这段代码的结果是 "6.8e+1"，前面解释过，它表示 6.8x10¹。问题是，如果不知道要用哪种形式（预定形式或指数形式）表示数字怎么办？可以用 toPrecision() 方法。

toPrecision() 方法

toPrecision() 方法根据最有意义的形式来返回数字的预定形式或指数形式。它有一个参数，即用于表示数的数字总数（不包括指数）。例如，

```
var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(1));  //输出 "7e+1"
```

这段代码的任务是用一位数字表示数字 68，结果为 "7e+1"，以另外的形式表示即 70。的确，toPrecision() 方法会对数进行舍入。不过，如果用 2 位数字表示 68，就容易多了：

```
var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(2));  //输出 "68"
```

当然，输出的是 "68"，因为这正是该数的准确表示。不过，如果指定的位数多于需要的位数又如何呢？

```
var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(3));  //输出 "68.0"
```

在这种情况下，toPrecision(3) 等价于 toFixed(1)，输出的是 "68.0"。

toFixed()、toExponential() 和 toPrecision() 方法都会进行舍入操作，以便用正确的小数位数正确地表示一个数。

提示：与 Boolean 对象相似，Number 对象也很重要，不过应该少用这种对象，以避免潜在的问题。只要可能，都使用数字的原始表示法。

参阅

如需更多有关 Number 对象的信息，请访问 JavaScript Number 对象参考手册。

String 对象

String 对象是 String 原始类型的对象表示法，它是以下方式创建的：

```
var oStringObject = new String("hello world");
```

String 对象的 valueOf() 方法和 toString() 方法都会返回 String 类型的原始值：

```
alert(oStringObject.valueOf() == oStringObject.toString());    //输出 "true"
```

如果运行这段代码，输出是 "true"，说明这些值真的相等。

注释：String 对象是 ECMAScript 中比较复杂的引用类型之一。同样，本节的重点只是 String 类的基本功能。更多的高级功能请阅读本教程相关的章节，或参阅 JavaScript String 对象参考手册。

length 属性

String 对象具有属性 length，它是字符串中的字符个数：

```
var oStringObject = new String("hello world");
alert(oStringObject.length);    //输出 "11"
```

这个例子输出的是 "11"，即 "hello world" 中的字符个数。注意，即使字符串包含双字节的字符（与 ASCII 字符相对，ASCII 字符只占用一个字节），每个字符也只算一个字符。

charAt() 和 charCodeAt() 方法

String 对象还拥有大量的方法。

首先，两个方法 charAt() 和 charCodeAt() 访问的是字符串中的单个字符。这两个方法都有一个参数，即要操作的字符的位置。

charAt() 方法返回的是包含指定位置处的字符的字符串：

```
var oStringObject = new String("hello world");
alert(oStringObject.charAt(1));    //输出 "e"
```

在字符串 "hello world" 中，位置 1 处的字符是 "e"。在“ECMAScript 原始类型”这一节中我们讲过，第一个字符的位置是 0，第二个字符的位置是 1，依此类推。因此，调用 charAt(1) 返回的是 "e"。

如果想得到的不是字符，而是字符代码，那么可以调用 charCodeAt() 方法：

```
var oStringObject = new String("hello world");
alert(oStringObject.charCodeAt(1));    //输出 "101"
```

这个例子输出 "101"，即小写字母 "e" 的字符代码。

concat() 方法

接下来是 concat() 方法，用于把一个或多个字符串连接到 String 对象的原始值上。该方法返回的是 String 原始值，保持原始的 String 对象不变：

```
var oStringObject = new String("hello ");
var sResult = oStringObject.concat("world");
alert(sResult);        //输出 "hello world"
alert(oStringObject);    //输出 "hello "
```

在上面这段代码中，调用 concat() 方法返回的是 "hello world"，而 String 对象存放的仍然是 "hello "。出于这种原因，较常见的是用加号（+）连接字符串，因为这种形式从逻辑上表明了真正的行为：

```
var oStringObject = new String("hello ");
var sResult = oStringObject + "world";
alert(sResult);        //输出 "hello world"
alert(oStringObject);    //输出 "hello "
```

indexOf() 和 lastIndexOf() 方法

迄今为止，已讨论过连接字符串的方法，访问字符串中的单个字符的方法。不过如果无法确定在某个字符串中是否确实存在一个字符，应该调用什么方法呢？这时，可调用 indexOf() 和 lastIndexOf() 方法。

indexOf() 和 lastIndexOf() 方法返回的都是指定的子串在另一个字符串中的位置，如果没有找不到子串，则返回 -1。

这两个方法的不同之处在于，indexOf() 方法是从字符串的开头（位置 0）开始检索字符串，而 lastIndexOf() 方法则是从字符串的结尾开始检索子串。例如：

```
var oStringObject = new String("hello world!");
alert(oStringObject.indexOf("o"));        输出 "4"
alert(oStringObject.lastIndexOf("o"));    输出 "7"
```

在这里，第一个 "o" 字符串出现在位置 4，即 "hello" 中的 "o"；最后一个 "o" 出现在位置 7，即 "world" 中的 "o"。如果该字符串中只有一个 "o" 字符串，那么 indexOf() 和 lastIndexOf() 方法返回的位置相同。

localeCompare() 方法

下一个方法是 localeCompare()，对字符串进行排序。该方法有一个参数 - 要进行比较的字符串，返回的是下列三个值之一：

- 如果 String 对象按照字母顺序排在参数中的字符串之前，返回负数。
- 如果 String 对象等于参数中的字符串，返回 0
- 如果 String 对象按照字母顺序排在参数中的字符串之后，返回正数。

注释：如果返回负数，那么最常见的是 -1，不过真正返回的是由实现决定的。如果返回正数，那么同样的，最常见的是 1，不过真正返回的是由实现决定的。

示例如下：

```
var oStringObject = new String("yellow");
alert(oStringObject.localeCompare("brick"));        //输出 "1"
alert(oStringObject.localeCompare("yellow"));        //输出 "0"
alert(oStringObject.localeCompare("zoo"));        //输出 "-1"
```

在这段代码中，字符串 "yellow" 与 3 个值进行了对比，即 "brick"、"yellow" 和 "zoo"。由于按照字母顺序排列，"yellow" 位于 "brick" 之后，所以 localeCompare() 返回 1；"yellow" 等于 "yellow"，所以 localeCompare() 返回 0；"zoo" 位于 "yellow" 之后，localeCompare() 返回 -1。再强调一次，由于返回的值是由实现决定的，所以最好以下面的方式调用 localeCompare() 方法：

```
var oStringObject1 = new String("yellow");
var oStringObject2 = new String("brick");

var iResult = oStringObject1.localeCompare(oStringObject2);

if(iResult < 0) {
  alert(oStringObject1 + " comes before " + oStringObject2);
} else if (iResult > 0) {
  alert(oStringObject1 + " comes after " + oStringObject2);
} else {
  alert("The two strings are equal");
}
```

采用这种结构，可以确保这段代码在所有实现中都能正确运行。

localeCompare() 方法的独特之处在于，实现所处的区域（locale，兼指国家/地区和语言）确切说明了这种方法运行的方式。在美国，英语是 ECMAScript 实现的标准语言，localeCompare() 是区分大小写的，大写字母在字母顺序上排在小写字母之后。不过，在其他区域，情况可能并非如此。

slice() 和 substring()

ECMAScript 提供了两种方法从子串创建字符串值，即 slice() 和 substring()。这两种方法返回的都是要处理的字符串的子串，都接受一个或两个参数。第一个参数是要获取的子串的起始位置，第二个参数（如果使用的话）是要获取子串终止前的位置（也就是说，获取终止位置处的字符不包括在返回的值内）。如果省略第二个参数，终止位就默认为字符串的长度。

与 concat() 方法一样，slice() 和 substring() 方法都不改变 String 对象自身的值。它们只返回原始的 String 值，保持 String 对象不变。

```
var oStringObject = new String("hello world");
alert(oStringObject.slice("3"));        //输出 "lo world"
alert(oStringObject.substring("3"));        //输出 "lo world"
alert(oStringObject.slice("3", "7"));        //输出 "lo w"
alert(oStringObject.substring("3", "7"));    //输出 "lo w"
```

在这个例子中，slice() 和 substring() 的用法相同，返回值也一样。当只有参数 3 时，两个方法返回的都是 "lo world"，因为 "hello" 中的第二个 "l" 位于位置 3 上。当有两个参数 "3" 和 "7" 时，两个方法返回的值都是 "lo w"（"world" 中的字母 "o" 位于位置 7 上，所以它不包括在结果中）。

为什么有两个功能完全相同的方法呢？事实上，这两个方法并不完全相同，不过只在参数为负数时，它们处理参数的方式才稍有不同。

对于负数参数，slice() 方法会用字符串的长度加上参数，substring() 方法则将其作为 0 处理（也就是说将忽略它）。例如：

```
var oStringObject = new String("hello world");
alert(oStringObject.slice("-3"));        //输出 "rld"
alert(oStringObject.substring("-3"));    //输出 "hello world"
alert(oStringObject.slice("3, -4"));        //输出 "lo w"
alert(oStringObject.substring("3, -4"));    //输出 "hel"
```

这样即可看出 slice() 和 substring() 方法的主要不同。

当只有参数 -3 时，slice() 返回 "rld"，substring() 则返回 "hello world"。这是因为对于字符串 "hello world"，slice("-3") 将被转换成 slice("8")，而 substring("-3") 将被转换成 substring("0")。

同样，使用参数 3 和 -4 时，差别也很明显。slice() 将被转换成 slice(3, 7)，与前面的例子相同，返回 "lo w"。而 substring() 方法则将两个参数解释为 substring(3, 0)，实际上即 substring(0, 3)，因为 substring() 总把较小的数字作为起始位，较大的数字作为终止位。因此，substring("3, -4") 返回的是 "hel"。这里的最后一行代码用来说明如何使用这些方法。

toLowerCase()、toLocaleLowerCase()、toUpperCase() 和 toLocaleUpperCase()

最后一套要讨论的方法涉及大小写转换。有 4 种方法用于执行大小写转换，即

- toLowerCase()
- toLocaleLowerCase()
- toUpperCase()
- toLocaleUpperCase()

从名字上可以看出它们的用途，前两种方法用于把字符串转换成全小写的，后两种方法用于把字符串转换成全大写的。

toLowerCase() 和 toUpperCase() 方法是原始的，是以 java.lang.String 中相同方法为原型实现的。

toLocaleLowerCase() 和 toLocaleUpperCase() 方法是基于特定的区域实现的（与 localeCompare() 方法相同）。在许多区域中，区域特定的方法都与通用的方法完全相同。不过，有几种语言对 Unicode 大小写转换应用了特定的规则（例如土耳其语），因此必须使用区域特定的方法才能进行正确的转换。

```
var oStringObject = new String("Hello World");
alert(oStringObject.toLocaleUpperCase());    //输出 "HELLO WORLD"
alert(oStringObject.toUpperCase());         //输出 "HELLO WORLD"
alert(oStringObject.toLocaleLowerCase());    //输出 "hello world"
alert(oStringObject.toLowerCase());         //输出 "hello world"
```

这段代码中，toUpperCase() 和 toLocaleUpperCase() 输出的都是 "HELLO WORLD"，toLowerCase() 和 toLocaleLowerCase() 输出的都是 "hello world"。一般来说，如果不知道在以哪种编码运行一种语言，则使用区域特定的方法比较安全。

提示：记住，String 对象的所有属性和方法都可应用于 String 原始值上，因为它们是伪对象。

instanceof 运算符

在使用 typeof 运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回 "object"。ECMAScript 引入了另一个 Java 运算符 instanceof 来解决这个问题。

instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。例如：

```
var oStringObject = new String("hello world");
alert(oStringObject instanceof String);    //输出 "true"
```

这段代码问的是“变量 oStringObject 是否为 String 对象的实例？”oStringObject 的确是 String 对象的实例，因此结果是 "true"。尽管不像 typeof 方法那样灵活，但是在 typeof 方法返回 "object" 的情况下，instanceof 方法还是很有用的。

ECMAScript 运算符

ECMAScript 一元运算符

一元运算符只有一个参数，即要操作的对象或值。它们是 **ECMAScript** 中最简单的运算符。

delete

delete 运算符删除对以前定义的对象属性或方法的引用。例如：

```
var o = new Object;
o.name = "David";
alert(o.name);    //输出 "David"
`delete o.name;`
alert(o.name);    //输出 "undefined"
```

在这个例子中，删除了 name 属性，意味着强制解除对它的引用，将其设置为 undefined（即创建的未初始化的变量的值）。

delete 运算符不能删除开发者未定义的属性和方法。例如，下面的代码将引发错误：

```
delete o.toString;
```

即使 toString 是有效的方法名，这行代码也会引发错误，因为 toString() 方法是原始的 ECMAScript 方法，不是开发者定义的。

void

void 运算符对任何值返回 undefined。该运算符通常用于避免输出不应该输出的值，例如，从 HTML 的 <a> 元素调用 JavaScript 函数时。要正确做到这一点，函数不能返回有效值，否则浏览器将清空页面，只显示函数的结果。例如：

```
<a href="javascript:window.open('about:blank')">Click me</a>
```

如果把这行代码放入 HTML 页面，点击其中的链接，即可看到屏幕上显示 "[object]"。TIY

这是因为 window.open() 方法返回了新打开的窗口的引用。然后该对象将被转换成要显示的字符串。

要避免这种效果，可以用 void 运算符调用 window.open() 函数：

```
<a href="javascript:void(window.open('about:blank'))">Click me</a>
```

这使 window.open() 调用返回 undefined，它不是有效值，不会显示在浏览器窗口中。

提示：请记住，没有返回值的函数真正返回的都是 undefined。

前增量/前减量运算符

直接从 C（和 Java）借用的两个运算符是前增量运算符和前减量运算符。

所谓前增量运算符，就是数值上加 1，形式是在变量前放两个加号（++）：

```
var iNum = 10;
++iNum;
```

第二行代码把 iNum 增加到了 11，它实质上等价于：

```
var iNum = 10;
iNum = iNum + 1;
```

同样，前减量运算符是从数值上减 1，形式是在变量前放两个减号（--）：

```
var iNum = 10;
--iNum;
```

在这个例子中，第二行代码把 iNum 的值减到 9。

在使用前缀式运算符时，注意增量和减量运算符都发生在计算表达式之前。考虑下面的例子：

```
var iNum = 10;
--iNum;
alert(iNum);    //输出 "9"
alert(--iNum);    //输出 "8"
alert(iNum);    //输出 "8"
```

第二行代码对 iNum 进行减量运算，第三行代码显示的结果是（"9"）。第四行代码又对 iNum 进行减量运算，不过这次前减量运算和输出操作出现在同一个语句中，显示的结果是 "8"。为了证明已实现了所有的减量操作，第五行代码又输出一次"8"。

在算术表达式中，前增量和前减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = --iNum1 + ++iNum2;    //等于 "22"
var iNum4 = iNum1 + iNum2;        //等于 "22"
```

在前面的代码中，iNum3 等于 22，因为表达式要计算的是 1 + 21。变量 iNum4 也等于 22，也是 1 + 21。

后增量/后减量运算符

还有两个直接从 C（和 Java）借用的运算符，即后增量运算符和后减量运算符。

后增量运算符也是给数值上加 1，形式是在变量后放两个加号（++）：

```
var iNum = 10;
iNum++;
```

不出所料，后减量运算符也是从数值上减 1，形式为在变量后加两个减号（--）：

```
var iNum = 10;
iNum--;
```

第二行代码把 iNum 的值减到 9。

与前缀式运算符不同的是，后缀式运算符是在计算过包含它们的表达式后才进行增量或减量运算的。考虑以下的例子：

```
var iNum = 10;
iNum--;
alert(iNum);    //输出 "9"
alert(iNum--);    //输出 "9"
alert(iNum);    //输出 "8"
```

与前缀式运算符的例子相似，第二行代码对 iNum 进行减量运算，第三行代码显示结果（"9"）。第四行代码继续显示 iNum 的值，不过这次是在同一语句中应用减量运算符。由于减量运算发生在计算过表达式之后，所以这条语句显示的数是 "9"。执行了第五行代码后，alert 函数显示的是 "8"，因为在执行第四行代码之后和执行第五行代码之前，执行了后减量运算。

在算术表达式中，后增量和后减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = iNum1-- + iNum2++;    //等于 "22"
var iNum4 = iNum1 + iNum2;        //等于 "22"
```

在前面的代码中，iNum3 等于 22，因为表达式要计算的是 2 + 20。变量 iNum4 也等于 22，不过它计算的是 1 + 21，因为增量和减量运算都在给 iNum3 赋值后才发生。

一元加法和一元减法

大多数人都熟悉一元加法和一元减法，它们在 ECMAScript 中的用法与您高中数学中学到的用法相同。

一元加法本质上对数字无任何影响：

```
var iNum = 20;
iNum = +iNum;
alert(iNum);    //输出 "20"
```

这段代码对数字 20 应用了一元加法，返回的还是 20。

尽管一元加法对数字无作用，但对字符串却有有趣的效果，会把字符串转换成数字。

```
var sNum = "20";
alert(typeof sNum);    //输出 "string"
var iNum = +sNum;
alert(typeof iNum);    //输出 "number"
```

这段代码把字符串 "20" 转换成真正的数字。当一元加法运算符对字符串进行操作时，它计算字符串的方式与 parseInt() 相似，主要的不同是只有对以 "0x" 开头的字符串（表示十六进制数字），一元运算符才能把它转换成十进制的值。因此，用一元加法转换 "010"，得到的总是 10，而 "0xB" 将被转换成 11。

另一方面，一元减法就是对数值求负（例如把 20 转换成 -20）：

```
var iNum = 20;
iNum = -iNum;
alert(iNum);    //输出 "-20"
```

与一元加法运算符相似，一元减法运算符也会把字符串转换成近似的数字，此外还会对该值求负。例如：

```
var sNum = "20";
alert(typeof sNum);    //输出 "string"
var iNum = -sNum;
alert(iNum);        //输出 "-20"
alert(typeof iNum);    //输出 "number"
```

在上面的代码中，一元减法运算符将把字符串 "-20" 转换成 -20（一元减法运算符对十六进制和十进制的处理方式与一元加法运算符相似，只是它还会对该值求负）。

ECMAScript 位运算符

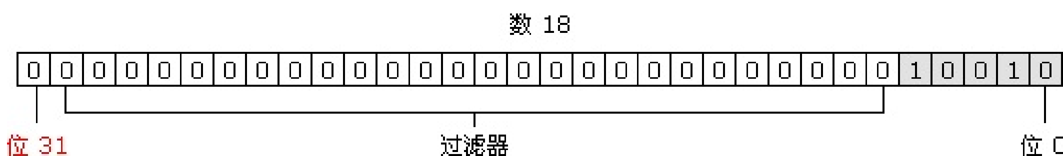
位运算符是在数字底层（即表示数字的 **32** 个数位）进行操作的。

重温整数

ECMAScript 整数有两种类型，即有符号整数（允许用正数和负数）和无符号整数（只允许用正数）。在 ECMAScript 中，所有整数字面量默认都是有符号整数，这意味着什么呢？

有符号整数使用 31 位表示整数的数值，用第 32 位表示整数的符号，0 表示正数，1 表示负数。数值范围从 -2147483648 到 2147483647。

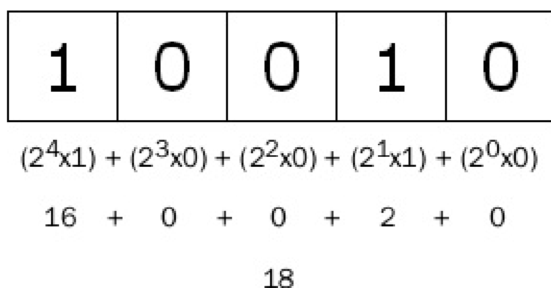
可以以两种不同的方式存储二进制形式的有符号整数，一种用于存储正数，一种用于存储负数。正数是以真二进制形式存储的，前 31 位中的每一位都表示 2 的幂，从第 1 位（位 0）开始，表示 2⁰，第 2 位（位 1）表示 2¹。没用到的位用 0 填充，即忽略不计。例如，下图展示的是数 18 的表示法。



18 的二进制版本只用了前 5 位，它们是这个数字的有效位。把数字转换成二进制字符串，就能看到有效位：

```
var iNum = 18;
alert(iNum.toString(2));    //输出 "10010"
```

这段代码只输出 "10010"，而不是 18 的 32 位表示。其他的数位并不重要，因为仅使用前 5 位即可确定这个十进制数值。如下图所示：



负数也存储为二进制代码，不过采用的形式是二进制补码。计算数字二进制补码的步骤有三步：

1. 确定该数字的非负版本的二进制表示（例如，要计算 -18的二进制补码，首先要确定 18 的二进制表示）

2. 求得二进制反码，即要把 0 替换为 1，把 1 替换为 0
3. 在二进制反码上加 1

要确定 -18 的二进制表示，首先必须得到 18 的二进制表示，如下所示：

```
0000 0000 0000 0000 0000 0000 0001 0010
```

接下来，计算二进制反码，如下所示：

```
1111 1111 1111 1111 1111 1111 1110 1101
```

最后，在二进制反码上加 1，如下所示：

```
1111 1111 1111 1111 1111 1111 1110 1101
                                       1
---------------------------------------
1111 1111 1111 1111 1111 1111 1110 1110
```

因此，-18 的二进制表示即 1111 1111 1111 1111 1111 1111 1110 1110。记住，在处理有符号整数时，开发者不能访问 31 位。

有趣的是，把负整数转换成二进制字符串后，ECMAScript 并不以二进制补码的形式显示，而是用数字绝对值的标准二进制代码前面加负号的形式输出。例如：

```
var iNum = -18;
alert(iNum.toString(2));    //输出 "-10010"
```

这段代码输出的是 "-10010"，而非二进制补码，这是为避免访问位 31。为了简便，ECMAScript 用一种简单的方式处理整数，使得开发者不必关心它们的用法。

另一方面，无符号整数把最后一位作为另一个数位处理。在这种模式中，第 32 位不表示数字的符号，而是值 2³¹。由于这个额外的位，无符号整数的数值范围为 0 到 4294967295。对于小于 2147483647 的整数来说，无符号整数看来与有符号整数一样，而大于 2147483647 的整数则要使用位 31（在有符号整数中，这一位总是 0）。

把无符号整数转换成字符串后，只返回它们的有效位。

注意：所有整数字面量都默认存储为有符号整数。只有 ECMAScript 的位运算符才能创建无符号整数。

位运算 NOT

位运算 NOT 由否定号（~）表示，它是 ECMAScript 中为数不多的与二进制算术有关的运算符之一。

位运算 NOT 是三步的处理过程：

1. 把运算数转换成 32 位数字
2. 把二进制数转换成它的二进制反码
3. 把二进制数转换成浮点数

例如：

```
var iNum1 = 25;        //25 等于 00000000000000000000000000011001
`var iNum2 = ~iNum1;`     //转换为 11111111111111111111111111100110
alert(iNum2);        //输出 "-26"
```

位运算 NOT 实质上是对数字求负，然后减 1，因此 25 变 -26。用下面的方法也可以得到同样的方法：

```
var iNum1 = 25;
var iNum2 = -iNum1 -1;
alert(iNum2);    //输出 -26
```

位运算 AND

位运算 AND 由和号（&）表示，直接对数字的二进制形式进行运算。它把每个数字中的数位对齐，然后用下面的规则对同一位置上的两个数位进行 AND 运算：

第一个数字中的数位	第二个数字中的数位	结果
1	1	1
1	0	0
0	1	0
0	0	0

例如，要对数字 25 和 3 进行 AND 运算，代码如下所示：

```
var iResult = 25 & 3;
alert(iResult);    //输出 "1"
```

25 和 3 进行 AND 运算的结果是 1。为什么？分析如下：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
---------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

可以看出，在 25 和 3 中，只有一个数位（位 0）存放的都是 1，因此，其他数位生成的都是 0，所以结果为 1。

位运算 OR

位运算 OR 由符号（|）表示，也是直接对数字的二进制形式进行运算。在计算每位时，OR 运算符采用下列规则：

第一个数字中的数位	第二个数字中的数位	结果
1	1	1
1	0	1
0	1	1
0	0	0

仍然使用 AND 运算符所用的例子，对 25 和 3 进行 OR 运算，代码如下：

```
var iResult = 25 | 3;
alert(iResult);    //输出 "27"
```

25 和 3 进行 OR 运算的结果是 27：

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

可以看出，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11011 等于 27。

位运算 XOR

位运算 XOR 由符号（^）表示，当然，也是直接对二进制形式进行运算。XOR 不同于 OR，当只有一个数位存放的是 1 时，它才返回 1。真值表如下：

第一个数字中的数位	第二个数字中的数位	结果
1	1	0
1	0	1
0	1	1
0	0	0

对 25 和 3 进行 XOR 运算，代码如下：

```
var iResult = 25 ^ 3;
alert(iResult);    //输出 "26"
```

25 和 3 进行 XOR 运算的结果是 26：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
---------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

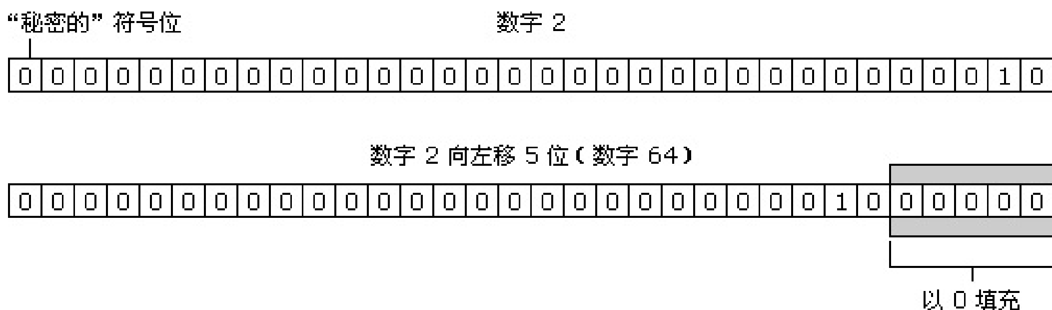
可以看出，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11010 等于 26。

左移运算

左移运算由两个小于号表示（<<）。它把数字中的所有数位向左移动指定的数量。例如，把数字 2（等于二进制中的 10）左移 5 位，结果为 64（等于二进制中的 1000000）：

```
var iOld = 2;        //等于二进制 10
var iNew = iOld << 5;    //等于二进制 1000000 十进制 64
```

注意：在左移数位时，数字右边多出 5 个空位。左移运算用 0 填充这些空位，使结果成为完整的 32 位数字。



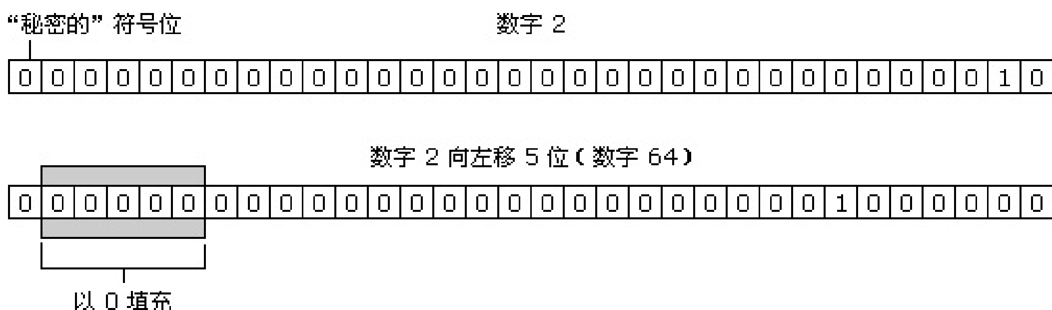
注意：左移运算保留数字的符号位。例如，如果把 -2 左移 5 位，得到的是 -64，而不是 64。“符号仍然存储在第 32 位中吗？”是的，不过这在 ECMAScript 后台进行，开发者不能直接访问第 32 个数位。即使输出二进制字符串形式的负数，显示的也是负号形式（例如，-2 将显示 -10。）

有符号右移运算

有符号右移运算符由两个大于号表示（>>）。它把 32 位数字中的所有数位整体右移，同时保留该数的符号（正号或负号）。有符号右移运算符恰好与左移运算相反。例如，把 64 右移 5 位，将变为 2：

```
var iOld = 64;        //等于二进制 1000000
var iNew = iOld >> 5;    //等于二进制 10 十进制 2
```

同样，移动数位后会造成空位。这次，空位位于数字的左侧，但位于符号位之后。ECMAScript 用符号位的值填充这些空位，创建完整的数字，如下图所示：



无符号右移运算

无符号右移运算符由三个大于号（>>>）表示，它将无符号 32 位数的所有数位整体右移。对于正数，无符号右移运算的结果与有符号右移运算一样。

用有符号右移运算中的例子，把 64 右移 5 位，将变为 2：

```
var iOld = 64;          //等于二进制 1000000
var iNew = iOld >>> 5;    //等于二进制 10 十进制 2
```

对于负数，情况就不同了。

无符号右移运算用 0 填充所有空位。对于正数，这与有符号右移运算的操作一样，而负数则被作为正数来处理。

由于无符号右移运算的结果是一个 32 位的正数，所以负数的无符号右移运算得到的总是一个非常大的数字。例如，如果把 -64 右移 5 位，将得到 134217726。如何得到这种结果的呢？

要实现这一点，需要把这个数字转换成无符号的等价形式（尽管该数字本身还是有符号的），可以通过以下代码获得这种形式：

```
var iUnsigned64 = -64 >>> 0;
```

然后，用 Number 类型的 toString() 获取它的真正的位表示，采用的基为 2：

```
alert(iUnsigned64.toString(2));
```

这将生成 11111111111111111111111111000000，即有符号整数 -64 的二进制补码表示，不过它等于无符号整数 4294967232。

出于这种原因，使用无符号右移运算符要小心。

ECMAScript Boolean 运算符

Boolean 运算符非常重要，它使得程序语言得以正常运行。

Boolean 运算符有三种：**NOT**、**AND** 和 **OR**。

ToBoolean 操作

在学习各种逻辑运算符之前，让我们先了解一下 ECMAScript-262 v5 规范中描述的 ToBoolean 操作。

抽象操作 ToBoolean 将其参数按照下表中的规则转换为逻辑值：

参数类型	结果
Undefined	false
Null	false
Boolean	结果等于输入的参数（不转换）
Number	如果参数为 +0, -0 或 NaN，则结果为 false；否则为 true。
String	如果参数为空字符串，则结果为 false；否则为 true。
Object	true

逻辑 NOT 运算符

在 ECMAScript 中，逻辑 NOT 运算符与 C 和 Java 中的逻辑 NOT 运算符相同，都由感叹号（!）表示。

与逻辑 OR 和逻辑 AND 运算符不同的是，逻辑 **NOT** 运算符返回的一定是 **Boolean** 值。

逻辑 NOT 运算符的行为如下：

- 如果运算数是对象，返回 false
- 如果运算数是数字 0，返回 true
- 如果运算数是 0 以外的任何数字，返回 false
- 如果运算数是 null，返回 true
- 如果运算数是 NaN，返回 true
- 如果运算数是 undefined，发生错误

通常，该运算符用于控制循环：

```
var bFound = false;
var i = 0;

`while (!bFound)` {
  if (aValue[i] == vSearchValues) {
    bFound = true;
  } else {
    i++;
  }
}
```

在这个例子中，Boolean 变量（bFound）用于记录检索是否成功。找到问题中的数据项时，bFound 将被设置为 true，!bFound 将等于 false，意味着运行将跳出 while 循环。

判断 ECMAScript 变量的 Boolean 值时，也可以使用逻辑 NOT 运算符。这样做需要在一行代码中使用两个 NOT 运算符。无论运算数是什么类型，第一个 NOT 运算符返回 Boolean 值。第二个 NOT 将对该 Boolean 值求负，从而给出变量真正的 Boolean 值。

```
var bFalse = false;
var sRed = "red";
var iZero = 0;
var iThreeFourFive = 345;
var oObject = new Object;

document.write("bFalse 的逻辑值是 " + (!!bFalse));
document.write("sRed 的逻辑值是 " + (!!sRed));
document.write("iZero 的逻辑值是 " + (!!iZero));
document.write("iThreeFourFive 的逻辑值是 " + (!!iThreeFourFive));
document.write("oObject 的逻辑值是 " + (!!oObject));
```

结果：

```
bFalse 的逻辑值是 false
sRed 的逻辑值是 true
iZero 的逻辑值是 false
iThreeFourFive 的逻辑值是 true
oObject 的逻辑值是 true
```

逻辑 AND 运算符

在 ECMAScript 中，逻辑 AND 运算符用双和号（&&）表示：

例如：

```
var bTrue = true;
var bFalse = false;
var bResult = bTrue && bFalse;
```

下面的真值表描述了逻辑 AND 运算符的行为：

运算数 1	运算数 2	结果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑 **AND** 运算的运算数可以是任何类型的，不止是 **Boolean** 值。

如果某个运算数不是原始的 Boolean 型值，逻辑 AND 运算并不一定返回 Boolean 值：

- 如果一个运算数是对象，另一个是 Boolean 值，返回该对象。
- 如果两个运算数都是对象，返回第二个对象。
- 如果某个运算数是 null，返回 null。
- 如果某个运算数是 NaN，返回 NaN。
- 如果某个运算数是 undefined，发生错误。

与 Java 中的逻辑 AND 运算相似，ECMAScript 中的逻辑 AND 运算也是简便运算，即如果第一个运算数决定了结果，就不再计算第二个运算数。对于逻辑 AND 运算来说，如果第一个运算数是 false，那么无论第二个运算数的值是什么，结果都不可能等于 true。

考虑下面的例子：

```
var bTrue = true;
var bResult = (bTrue && bUnknown);    //发生错误
alert(bResult);             //这一行不会执行
```

这段代码在进行逻辑 AND 运算时将引发错误，因为变量 bUnknown 是未定义的。变量 bTrue 的值为 true，因为逻辑 AND 运算将继续计算变量 bUnknown。这样做就会引发错误，因为 bUnknown 的值是 undefined，不能用于逻辑 AND 运算。

如果修改这个例子，把第一个数设为 false，那么就不会发生错误：

```
var bFalse = false;
var bResult = (bFalse && bUnknown);
alert(bResult);             //输出 "false"
```

在这段代码中，脚本将输出逻辑 AND 运算返回的值，即字符串 "false"。即使变量 bUnknown 的值为 undefined，它也不会被计算，因为第一个运算数的值是 false。

提示：在使用逻辑 AND 运算符时，必须记住它的这种简便计算特性。

逻辑 OR 运算符

ECMAScript 中的逻辑 OR 运算符与 Java 中的相同，都由双竖线（||）表示：

```
var bTrue = true;
var bFalse = false;
var bResult = bTrue || bFalse;
```

下面的真值表描述了逻辑 OR 运算符的行为：

运算数 1	运算数 2	结果
true	true	true
true	false	true
false	true	true
false	false	false

与逻辑 AND 运算符相似，如果某个运算数不是 Boolean 值，逻辑 OR 运算并不一定返回 Boolean 值：

- 如果一个运算数是对象，并且该对象左边的运算数值均为 false，则返回该对象。
- 如果两个运算数都是对象，返回第一个对象。
- 如果最后一个运算数是 null，并且其他运算数值均为 false，则返回 null。
- 如果最后一个运算数是 NaN，并且其他运算数值均为 false，则返回 NaN。
- 如果某个运算数是 undefined，发生错误。

与逻辑 AND 运算符一样，逻辑 OR 运算也是简便运算。对于逻辑 OR 运算符来说，如果第一个运算数值为 true，就不再计算第二个运算数。

例如：

```
var bTrue = true;
var bResult = (bTrue || bUnknown);
alert(bResult);            //输出 "true"
```

与前面的例子相同，变量 bUnknown 是未定义的。不过，由于变量 bTrue 的值为 true，bUnknown 不会被计算，因此输出的是 "true"。

如果把 bTrue 改为 false，将发生错误：

```
var bFalse = false;
var bResult = (bFalse || bUnknown);    //发生错误
alert(bResult);            //不会执行这一行
```

ECMAScript 乘性运算符

ECMAScript 的乘性运算符与 **Java**、**C**、**Perl** 等于语言中的同类运算符的运算方式相似。

需要注意的是，乘性运算符还具有一些自动转换功能。

乘法运算符

乘法运算符由星号（*）表示，用于两数相乘。

ECMAScript 中的乘法语法与 C 语言中的相同：

```
var iResult = 12 * 34
```

不过，在处理特殊值时，ECMAScript 中的乘法还有一些特殊行为：

- 如果结果太大或太小，那么生成的结果是 Infinity 或 -Infinity。
- 如果某个运算数是 NaN，结果为 NaN。
- Infinity 乘以 0，结果为 NaN。
- Infinity 乘以 0 以外的任何数字，结果为 Infinity 或 -Infinity。
- Infinity 乘以 Infinity，结果为 Infinity。

注释：如果运算数是数字，那么执行常规的乘法运算，即两个正数或两个负数为正数，两个运算数符号不同，结果为负数。

除法运算符

除法运算符由斜杠（/）表示，用第二个运算数除第一个运算数：

```
var iResult = 88 /11;
```

与乘法运算符相似，在处理特殊值时，除法运算符也有一些特殊行为：

- 如果结果太大或太小，那么生成的结果是 Infinity 或 -Infinity。
- 如果某个运算数是 NaN，结果为 NaN。
- Infinity 被 Infinity 除，结果为 NaN。
- Infinity 被任何数字除，结果为 Infinity。
- 0 除一个任何非无穷大的数字，结果为 NaN。
- Infinity 被 0 以外的任何数字除，结果为 Infinity 或 -Infinity。

注释：如果运算数是数字，那么执行常规的除法运算，即两个正数或两个负数为正数，两个运算数符号不同，结果为负数。

取模运算符

除法（余数）运算符由百分号（%）表示，使用方法如下：

```
var iResult = 26%5; //等于 1
```

与其他乘性运算符相似，对于特殊值，取模运算符也有特殊的行为：

- 如果被除数是 Infinity，或除数是 0，结果为 NaN。
- Infinity 被 Infinity 除，结果为 NaN。
- 如果除数是无穷大的数，结果为被除数。
- 如果被除数为 0，结果为 0。

注释：如果运算数是数字，那么执行常规的算术除法运算，返回除法运算得到的余数。

ECMAScript 加性运算符

在多数程序设计语言中，加性运算符（即加号或减号）通常是最简单的数学运算符。

在 **ECMAScript** 中，加性运算符有大量的特殊行为。

加法运算符

法运算符由加号（+）表示：

```
var iResult = 1 + 2
```

与乘性运算符一样，在处理特殊值时，ECMAScript 中的加法也有一些特殊行为：

- 某个运算数是 NaN，那么结果为 NaN。
- -Infinity 加 -Infinity，结果为 -Infinity。
- Infinity 加 -Infinity，结果为 NaN。
- +0 加 +0，结果为 +0。
- -0 加 +0，结果为 +0。
- -0 加 -0，结果为 -0。

不过，如果某个运算数是字符串，那么采用下列规则：

- 如果两个运算数都是字符串，把第二个字符串连接到第一个上。
- 如果只有一个运算数是字符串，把另一个运算数转换成字符串，结果是两个字符串连接成的字符串。

例如：

```
var result = 5 + 5;    //两个数字
alert(result);        //输出 "10"
var result2 = 5 + "5";    //一个数字和一个字符串
alert(result);        //输出 "55"
```

这段代码说明了加法运算符的两种模式之间的差别。正常情况下，5+5 等于 10（原始数值），如上述代码中前两行所示。不过，如果把一个运算数改为字符串 "5"，那么结果将变为 "55"（原始的字符串值），因为另一个运算数也会被转换为字符串。

注意：为了避免 JavaScript 中的一种常见错误，在使用加法运算符时，一定要仔细检查运算数的数据类型。

减法运算符

减法运算符（-），也是一个常用的运算符：

```
var iResult = 2 - 1;
```

与加法运算符一样，在处理特殊值时，减法运算符也有一些特殊行为：

- 某个运算数是 NaN，那么结果为 NaN。
- Infinity 减 Infinity，结果为 NaN。
- -Infinity 减 -Infinity，结果为 NaN。
- Infinity 减 -Infinity，结果为 Infinity。
- -Infinity 减 Infinity，结果为 -Infinity。
- +0 减 +0，结果为 +0。
- -0 减 -0，结果为 -0。
- +0 减 -0，结果为 +0。
- 某个运算符不是数字，那么结果为 NaN。

注释：如果运算数都是数字，那么执行常规的减法运算，并返回结果。

ECMAScript 关系运算符

关系运算符执行的是比较运算。每个关系运算符都返回一个布尔值。

常规比较方式

关系运算符小于、大于、小于等于和大于等于执行的是两个数的比较运算，比较方式与算术比较运算相同。

每个关系运算符都返回一个布尔值：

```
var bResult1 = 2 > 1    //true
var bResult2 = 2 < 1    //false
```

不过，对两个字符串应用关系运算符，它们的行为则不同。许多人认为小于表示“在字母顺序上靠前”，大于表示“在字母顺序上靠后”，但事实并非如此。对于字符串，第一个字符串中每个字符的代码都与会第二个字符串中对应位置的字符的代码进行数值比较。完成这种比较操作后，返回一个 Boolean 值。问题在于大写字母的代码都小于小写字母的代码，这意味这着可能会遇到下列情况：

```
var bResult = "Blue" < "alpha";
alert(bResult);    //输出 true
```

在上面的例子中，字符串 "Blue" 小于 "alpha"，因为字母 B 的字符代码是 66，字母 a 的字符代码是 97。要强制性得到按照真正的字母顺序比较的结果，必须把两个数转换成相同的大小写形式（全大写或全小写的），然后再进行比较：

```
var bResult = "Blue".toLowerCase() < "alpha".toLowerCase();
alert(bResult);    //输出 false
```

把两个运算数都转换成小写，确保了正确识别出 "alpha" 在字母顺序上位于 "Blue" 之前。

比较数字和字符串

另一种棘手的状况发生在比较两个字符串形式的数字时，比如：

```
var bResult = "25" < "3";
alert(bResult);    //输出 "true"
```

上面这段代码比较的是字符串 "25" 和 "3"。两个运算数都是字符串，所以比较的是它们的字符代码（"2" 的字符代码是 50，"3" 的字符代码是 51）。

不过，如果把某个运算数该为数字，那么结果就有趣了：

```
var bResult = "25" < 3;
alert(bResult);    //输出 "false"
```

这里，字符串 "25" 将被转换成数字 25，然后与数字 3 进行比较，结果不出所料。

无论何时比较一个数字和一个字符串，ECMAScript 都会把字符串转换成数字，然后按照数字顺序比较它们。

不过，如果字符串不能转换成数字又该如何呢？考虑下面的例子：

```
var bResult = "a" < 3;
alert(bResult);
```

你能预料到这段代码输出什么吗？字母 "a" 不能转换成有意义的数字。不过，如果对它调用 parseInt() 方法，返回的是 NaN。根据规则，任何包含 NaN 的关系运算符都要返回 false，因此这段代码也输出 false：

```
var bResult = "a" >= 3;
alert(bResult);
```

通常，如果小于运算的两个值返回 false，那么大于等于运算必须返回 true，不过如果某个数字是 NaN，情况则非如此。

ECMAScript 等性运算符

判断两个变量是否相等是程序设计中非常重要的运算。在处理原始值时，这种运算相当简单，但涉及对象，任务就稍有点复杂。

ECMAScript 提供了两套等性运算符：等号和非等号用于处理原始值，全等号和非全等号用于处理对象。

等号和非等号

在 ECMAScript 中，等号由双等号（==）表示，当且仅当两个运算数相等时，它返回 true。非等号由感叹号加等号（!=）表示，当且仅当两个运算数不相等时，它返回 true。为确定两个运算数是否相等，这两个运算符都会进行类型转换。

执行类型转换的规则如下：

- 如果一个运算数是 Boolean 值，在检查相等性之前，把它转换成数字值。false 转换成 0，true 为 1。
- 如果一个运算数是字符串，另一个是数字，在检查相等性之前，要尝试把字符串转换成数字。
- 如果一个运算数是对象，另一个是字符串，在检查相等性之前，要尝试把对象转换成字符串。
- 如果一个运算数是对象，另一个是数字，在检查相等性之前，要尝试把对象转换成数字。

在比较时，该运算符还遵守下列规则：

- 值 null 和 undefined 相等。
- 在检查相等性时，不能把 null 和 undefined 转换成其他值。
- 如果某个运算数是 NaN，等号将返回 false，非等号将返回 true。
- 如果两个运算数都是对象，那么比较的是它们的引用值。如果两个运算数指向同一对象，那么等号返回 true，否则两个运算数不等。

重要提示：即使两个数都是 NaN，等号仍然返回 false，因为根据规则，NaN 不等于 NaN。

下表列出了一些特殊情况，以及它们的结果：

表达式	值
null == undefined	true
"NaN" == NaN	false
5 == NaN	false
NaN == NaN	false
NaN != NaN	true
false == 0	true
true == 1	true
true == 2	false
undefined == 0	false
null == 0	false
"5" == 5	true

全等号和非全等号

等号和非等号的同类运算符是全等号和非全等号。这两个运算符所做的与等号和非等号相同，只是它们在检查相等性前，不执行类型转换。

全等号由三个等号表示（===），只有在无需类型转换运算数就相等的情况下，才返回 true。

例如：

```
var sNum = "66";
var iNum = 66;
alert(sNum == iNum);    //输出 "true"
alert(sNum === iNum);    //输出 "false"
```

在这段代码中，第一个 alert 使用等号来比较字符串 "66" 和数字 66，输出 "true"。如前所述，这是因为字符串 "66" 将被转换成数字 66，，然后才与另一个数字 66 进行比较。第二个 alert 使用全等号在没有类型转换的情况下比较字符串和数字，当然，字符串不等于数字，所以输出 "false"。

非全等号由感叹号加两个等号（!==）表示，只有在无需类型转换运算数不相等的情况下，才返回 true。

例如：

```
var sNum = "66";
var iNum = 66;
alert(sNum != iNum);    //输出 "false"
alert(sNum !== iNum);    //输出 "true"
```

这里，第一个 alert 使用非等号，把字符串 "66" 转换成数字 66，使得它与第二个运算数 66 相等。因此，计算结果为 "false"，因为两个运算数是相等的。第二个 alert 使用的非全等号。该运算是在问："sNum" 与 "iNum" 不同吗？这个问题的答案是：是的（true），因为 sNum 是字符串，而 iNum 是数字，它们当然不同。

ECMAScript 条件运算符

条件运算符

条件运算符是 ECMAScript 中功能最多的运算符，它的形式与 Java 中的相同。

```
_variable = boolean_expression ? true_value : false_value_;
```

该表达式主要是根据 *boolean_expression* 的计算结果有条件地为变量赋值。如果 *Boolean_expression* 为 true，就把 *true_value* 赋给变量；如果它是 false，就把 *false_value* 赋给变量。

例如：

```
var iMax = (iNum1 > iNum2) ? iNum1 : iNum2;
```

在这里例子中，iMax 将被赋予数字中的最大值。表达式声明如果 iNum1 大于 iNum2，则把 iNum1 赋予 iMax。但如果表达式为 false（即 iNum2 大于或等于 iNum1），则把 iNum2 赋予 iMax。

ECMAScript 赋值运算符

赋值运算符

简单的赋值运算符由等号（=）实现，只是把等号右边的值赋予等号左边的变量。

例如：

```
var iNum = 10;
```

复合赋值运算是由乘性运算符、加性运算符或位移运算符加等号（=）实现的。这些赋值运算符是下列这些常见情况的缩写形式：

```
var iNum = 10;
iNum = iNum + 10;
```

可以用一个复合赋值运算符改写第二行代码：

```
var iNum = 10;
iNum += 10;
```

每种主要的算术运算以及其他几个运算都有复合赋值运算符：

- 乘法/赋值（ `*=` ）
- 除法/赋值（ `/=` ）
- 取模/赋值（ `%=` ）
- 加法/赋值（ `+=` ）
- 减法/赋值（ `-=` ）
- 左移/赋值（ `<<=` ）
- 有符号右移/赋值（ `>>=` ）
- 无符号右移/赋值（ `>>>=` ）

ECMAScript 逗号运算符

逗号运算符

用逗号运算符可以在一条语句中执行多个运算。

例如：

```
var iNum1 = 1, iNum = 2, iNum3 = 3;
```

逗号运算符常用变量声明中。

ECMAScript 语句

ECMAScript if 语句

if 语句是 ECMAScript 中最常用的语句之一。

ECMAScript 语句

ECMA - 262 描述了 ECMAScript 的几种语句（statement）。

语句主要定义了 ECMAScript 的大部分语句，通常采用一个或多个关键字，完成给定的任务。

语句可以非常简单，例如通知函数退出，也可以非常复杂，如声明一组要反复执行的命令。

在《ECMAScript 语句》这一章，我们介绍了所有标准的 ECMAScript 语句。

if 语句

if 语句是 ECMAScript 中最常用的语句之一，事实上在许多计算机语言中都是如此。

if 语句的语法：

```
`if` (_condition_) _statement1_ `else` _statement2_
```

其中 *condition* 可以是任何表达式，计算的结果甚至不必是真正的 boolean 值，ECMAScript 会把它转换成 boolean 值。

如果条件计算结果为 true，则执行 *statement1*；如果条件计算结果为 false，则执行 *statement2*。

每个语句都可以是单行代码，也可以是代码块。

例如：

```
if (i > 30)
  {alert("大于 30");}
else
  {alert("小于等于 30");}
```

提示：使用代码块被认为是一种最佳的编程实践，即使要执行的代码只有一行。这样做可以使每个条件要执行什么一目了然。

还可以串联多个 if 语句。就像这样：

```
`if` (_condition1_) _statement1_ `else if` (_condition2_) _statement2_ `else` _statement3
```

例如：

```
if (i > 30) {
  alert("大于 30");
} else if (i < 0) {
  alert("小于 0");
} else {
  alert("在 0 到 30 之间");
}
```

ECMAScript 迭代语句

迭代语句又叫循环语句，声明一组要反复执行的命令，直到满足某些条件为止。

循环通常用于迭代数组的值（因此而得名），或者执行重复的算术任务。

本节为您介绍 **ECMAScript** 提供的四种迭代语句。

do-while 语句

do-while 语句是后测试循环，即退出条件在执行循环内部的代码之后计算。这意味着在计算表达式之前，至少会执行循环主体一次。

它的语法如下：

```
`do` {_statement_} `while` (_expression_);
```

例子：

```
var i = 0;
do {i += 2;} while (i < 10);
```

while 语句

while 语句是前测试循环。这意味着退出条件是在执行循环内部的代码之前计算的。因此，循环主体可能根本不被执行。

它的语法如下：

```
`while` (_expression_) _statement_
```

例子：

```
var i = 0;
while (i < 10) {
  i += 2;
}
```

for 语句

for 语句是前测试循环，而且在进入循环之前，能够初始化变量，并定义循环后要执行的代码。

它的语法如下：

```
`for` (_initialization; expression; post-loop-expression_) _statement_
```

注意：*post-loop-expression* 之后不能写分号，否则无法运行。

例子：

```
iCount = 6;
for (var i = 0; i < iCount; i++) {
  alert(i);
}
```

这段代码定义了初始值为 0 的变量 i。只有当条件表达式（i < iCount）的值为 true 时，才进入 for 循环，这样循环主体可能不被执行。如果执行了循环主体，那么将执行循环后表达式，并迭代变量 i。

for-in 语句

for 语句是严格的迭代语句，用于枚举对象的属性。

它的语法如下：

```
`for` (_property_ `in` _expression_) _statement_
```

例子：

```
for (sProp in window) {
  alert(sProp);
}
```

这里，for-in 语句用于显示 window 对象的所有属性。

前面讨论过的 PropertyIsEnumerable() 是 ECMAScript 中专门用于说明属性是否可以用 for-in 语句访问的方法。

ECMAScript 标签语句

有标签的语句

可以用下列语句给语句加标签，以便以后调用：

```
_label_ : _statement_
```

例如：

```
start : i = 5;
```

在这个例子中，标签 start 可以被之后的 break 或 continue 语句引用。

提示：在下面的章节，我们将为您介绍 break 和 continue 语句。

ECMAScript break 和 continue 语句

break 和 continue 语句对循环中的代码执行提供了更严格的控制。

break 和 continue 语句的不同之处

break 语句可以立即退出循环，阻止再次反复执行任何代码。

而 continue 语句只是退出当前循环，根据控制表达式还允许继续进行下一次循环。

例如：

```
var iNum = 0;

for (var i=1; i<10; i++) {
  if (i % 5 == 0) {
    `break;`
  }
  iNum++;
}
alert(iNum);    //输出 "4"
```

在以上代码中，for 循环从 1 到 10 迭代变量 i。在循环主体中，if 语句将（使用取模运算符）检查 i 的值是否能被 5 整除。如果能被 5 整除，将执行 break 语句。alert 显示 "4"，即退出循环前执行循环的次数。

如果用 continue 语句代替这个例子中的 break 语句，结果将不同：

```
var iNum = 0;

for (var i=1; i<10; i++) {
  if (i % 5 == 0) {
    `continue;`
  }
  iNum++;
}
alert(iNum);    //输出 "8"
```

这里，alert 将显示 "8"，即执行循环的次数。可能执行的循环总数为 9，不过当 i 的值为 5 时，将执行 continue 语句，会使循环跳过表达式 iNum++，返回循环开头。

与有标签的语句一起使用

break 语句和 continue 语句都可以与有标签的语句联合使用，返回代码中的特定位置。

通常，当循环内部还有循环时，会这样做，例如：

```
var iNum = 0;

outermost:
for (var i=0; i<10; i++) {
  for (var j=0; j<10; j++) {
    if (i == 5 && j == 5) {
    `break outermost;`
  }
  iNum++;
  }
}

alert(iNum);    //输出 "55"
```

在上面的例子中，标签 outermost 表示的是第一个 for 语句。正常情况下，每个 for 语句执行 10 次代码块，这意味着 iNum++ 正常情况下将被执行 100 次，在执行完成时，iNum 应该等于 100。这里的 break 语句有一个参数，即停止循环后要跳转到的语句的标签。这样 break 语句不止能跳出内部 for 语句（即使用变量 j 的语句），还能跳出外部 for 语句（即使用变量 i 的语句）。因此，iNum 最后的值是 55，因为当 i 和 j 的值都等于 5 时，循环将终止。

可以以相同的方式使用 continue 语句：

```
var iNum = 0;

outermost:
for (var i=0; i<10; i++) {
  for (var j=0; j<10; j++) {
    if (i == 5 && j == 5) {
    `continue outermost;`
  }
  iNum++;
  }
}

alert(iNum);    //输出 "95"
```

在上例中，continue 语句会迫使循环继续，不止是内部循环，外部循环也如此。当 j 等于 5 时出现这种情况，意味着内部循环将减少 5 次迭代，致使 iNum 的值为 95。

提示：可以看出，与 break 和 continue 联合使用的有标签语句非常强大，不过过度使用它们会给调试代码带来麻烦。要确保使用的标签具有说明性，同时不要嵌套太多层循环。

提示：想了解什么是有标签语句，请阅读 ECMAScript 标签语句 这一节。

ECMAScript with 语句

有标签的语句

with 语句用于设置代码在特定对象中的作用域。

它的语法：

```
with (_expression_) _statement_
```

例如：

```
var sMessage = "hello";
with(sMessage) {
  alert(toUpperCase());    //输出 "HELLO"
}
```

在这个例子中，with 语句用于字符串，所以在调用 toUpperCase() 方法时，解释程序将检查该方法是否是本地函数。如果不是，它将检查伪对象 sMessage，看它是否为该对象的方法。然后，alert 输出 "HELLO"，因为解释程序找到了字符串 "hello" 的 toUpperCase() 方法。

提示：with 语句是运行缓慢的代码块，尤其是在已设置了属性值时。大多数情况下，如果可能，最好避免使用它。

ECMAScript switch 语句

switch 语句

switch 语句是 if 语句的兄弟语句。

开发者可以用 switch 语句为表达式提供一系列的情况（case）。

switch 语句的语法：

```
switch (_expression_)
  case _value_: _statement_;
    break;
  case _value_: _statement_;
    break;
  case _value_: _statement_;
    break;
  case _value_: _statement_;
    break;
...
  case _value_: _statement_;
    break;
  default: _statement_;
```

每个情况（case）都是表示“如果 *expression* 等于 *value*，就执行 *statement*”。

关键字 break 会使代码跳出 switch 语句。如果没有关键字 break，代码执行就会继续进入下一个 case。

关键字 default 说明了表达式的结果不等于任何一种情况时的操作（事实上，它相对于 else 从句）。

switch 语句主要是为避免让开发者编写下面的代码：

```
if (i == 20)
  alert("20");
else if (i == 30)
  alert("30");
else if (i == 40)
  alert("40");
else
  alert("other");
```

等价的 switch 语句是这样的：

```
switch (i) {
  case 20: alert("20");
    break;
  case 30: alert("30");
    break;
  case 40: alert("40");
    break;
  default: alert("other");
}
```

ECMAScript 和 Java 中的 switch 语句

ECMAScript 和 Java 中的 switch 语句有两点不同。在 ECMAScript 中，switch 语句可以用于字符串，而且能用不是常量的值说明情况：

```
var BLUE = "blue", RED = "red", GREEN  = "green";

switch (sColor) {
  case BLUE: alert("Blue");
    break;
  case RED: alert("Red");
    break;
  case GREEN: alert("Green");
    break;
  default: alert("Other");
}
```

这里，switch 语句用于字符串 sColor，声明 case 使用的是变量 BLUE、RED 和 GREEN，这在 ECMAScript 中是完全有效的。

ECMAScript 函数

ECMAScript 函数概述

什么是函数？

函数是一组可以随时随地运行的语句。

函数是 ECMAScript 的核心。

函数是由这样的方式进行声明的：关键字 function、函数名、一组参数，以及置于括号中的待执行代码。

函数的基本语法是这样的：

```
function _functionName(arg0, arg1, ... argN)_ {
  _statements_
}
```

例如：

```
function sayHi(sName, sMessage) {
  alert("Hello " + sName + sMessage);
}
```

如何调用函数？

函数可以通过其名字加上括号中的参数进行调用，如果有多个参数。

如果您想调用上例中的那个函数，可以使用如下的代码：

```
sayHi("David", " Nice to meet you!")
```

调用上面的函数 sayHi() 会生成一个警告窗口。您可以亲自试一试这个例子。

函数如何返回值？

函数 sayHi() 未返回值，不过不必专门声明它（像在 Java 中使用 void 那样）。

即使函数确实有值，也不必明确地声明它。该函数只需要使用 return 运算符后跟要返回的值即可。

```
function sum(iNum1, iNum2) {
  return iNum1 + iNum2;
}
```

下面的代码把 sum 函数返回的值赋予一个变量：

```
var iResult = sum(1,1);
alert(iResult);    //输出 "2"
```

另一个重要概念是，与在 Java 中一样，函数在执行过 return 语句后立即停止代码。因此，return 语句后的代码都不会被执行。

例如，在下面的代码中，alert 窗口就不会显示出来：

```
function sum(iNum1, iNum2) {
  return iNum1 + iNum2;
  alert(iNum1 + iNum2);
}
```

一个函数中可以有多个 return 语句，如下所示：

```
function diff(iNum1, iNum2) {
  if (iNum1 > iNum2) {
    return iNum1 - iNum2;
  } else {
    return iNum2 - iNum1;
  }
}
```

上面的函数用于返回两个数的差。要实现这一点，必须用较大的数减去较小的数，因此用 if 语句决定执行哪个 return 语句。

如果函数无返回值，那么可以调用没有参数的 return 运算符，随时退出函数。

例如：

```
function sayHi(sMessage) {
  if (sMessage == "bye") {
    return;
  }

  alert(sMessage);
}
```

这段代码中，如果 sMessage 等于 "bye"，就永远不显示警告框。

注释：如果函数无明确的返回值，或调用了没有参数的 return 语句，那么它真正返回的值是 undefined。

ECMAScript arguments 对象

arguments 对象

在函数代码中，使用特殊对象 arguments，开发者无需明确指出参数名，就能访问它们。

例如，在函数 sayHi() 中，第一个参数是 message。用 arguments[0] 也可以访问这个值，即第一个参数的值（第一个参数位于位置 0，第二个参数位于位置 1，依此类推）。

因此，无需明确命名参数，就可以重写函数：

```
function sayHi() {
  if (`arguments[0]` == "bye") {
    return;
  }

  alert(arguments[0]);
}
```

检测参数个数

还可以用 arguments 对象检测函数的参数个数，引用属性 arguments.length 即可。

下面的代码将输出每次调用函数使用的参数个数：

```
function howManyArgs() {
  alert(arguments.length);
}

howManyArgs("string", 45);
howManyArgs();
howManyArgs(12);
```

上面这段代码将依次显示 "2"、"0" 和 "1"。

注释：与其他程序设计语言不同，ECMAScript 不会验证传递给函数的参数个数是否等于函数定义的参数个数。开发者定义的函数都可以接受任意个数的参数（根据 Netscape 的文档，最多可接受 255 个），而不会引发任何错误。任何遗漏的参数都会以 undefined 传递给函数，多余的函数将忽略。

模拟函数重载

用 arguments 对象判断传递给函数的参数个数，即可模拟函数重载：

```
function doAdd() {
  if(arguments.length == 1) {
    alert(arguments[0] + 5);
  } else if(arguments.length == 2) {
    alert(arguments[0] + arguments[1]);
  }
}

doAdd(10);    //输出 "15"
doAdd(40, 20);    //输出 "60"
```

当只有一个参数时，doAdd() 函数给参数加 5。如果有两个参数，则会把两个参数相加，返回它们的和。所以，doAdd(10) 输出的是 "15"，而 doAdd(40, 20) 输出的是 "60"。

虽然不如重载那么好，不过已足以避开 ECMAScript 的这种限制。

ECMAScript Function 对象（类）

ECMAScript 的函数实际上是功能完整的对象。

Function 对象（类）

ECMAScript 最令人感兴趣的可能莫过于函数实际上是功能完整的对象。

Function 类可以表示开发者定义的任何函数。

用 Function 类直接创建函数的语法如下：

```
var function_name = new function(_arg1_, _arg2_, ..., _argN_, _function_body_)
```

在上面的形式中，每个 *arg* 都是一个参数，最后一个参数是函数主体（要执行的代码）。这些参数必须是字符串。

记得下面这个函数吗？

```
function sayHi(sName, sMessage) {
  alert("Hello " + sName + sMessage);
}
```

还可以这样定义它：

```
var sayHi
=
new Function("sName", "sMessage", "alert(\"Hello \" + sName + sMessage);");
```

虽然由于字符串的关系，这种形式写起来有些困难，但有助于理解函数只不过是一种引用类型，它们的行为与用 Function 类明确创建的函数行为是相同的。

请看下面这个例子：

```
function doAdd(iNum) {
  alert(iNum + 20);
}

function doAdd(iNum) {
  alert(iNum + 10);
}

doAdd(10);    //输出 "20"
```

如你所知，第二个函数重载了第一个函数，使 doAdd(10) 输出了 "20"，而不是 "30"。

如果以下面的形式重写该代码块，这个概念就清楚了：

```
var doAdd = new Function("iNum", "alert(iNum + 20)");
var doAdd = new Function("iNum", "alert(iNum + 10)");
doAdd(10);
```

请观察这段代码，很显然，doAdd 的值被改成了指向不同对象的指针。函数名只是指向函数对象的引用值，行为就像其他对象一样。甚至可以使两个变量指向同一个函数：

```
var doAdd = new Function("iNum", "alert(iNum + 10)");
var alsodoAdd = doAdd;
doAdd(10);    //输出 "20"
alsodoAdd(10);    //输出 "20"
```

在这里，变量 doAdd 被定义为函数，然后 alsodoAdd 被声明为指向同一个函数的指针。用这两个变量都可以执行该函数的代码，并输出相同的结果 - "20"。因此，如果函数名只是指向函数的变量，那么可以把函数作为参数传递给另一个函数吗？回答是肯定的！

```
function callAnotherFunc(fnFunction, vArgument) {
  fnFunction(vArgument);
}

var doAdd = new Function("iNum", "alert(iNum + 10)");

callAnotherFunc(doAdd, 10);    //输出 "20"
```

在上面的例子中，callAnotherFunc() 有两个参数 - 要调用的函数和传递给该函数的参数。这段代码把 doAdd() 传递给 callAnotherFunc() 函数，参数是 10，输出 "20"。

注意：尽管可以使用 Function 构造函数创建函数，但最好不要使用它，因为用它定义函数比用传统方式要慢得多。不过，所有函数都应看作 Function 类的实例。

Function 对象的 length 属性

如前所述，函数属于引用类型，所以它们也有属性和方法。

ECMAScript 定义的属性 length 声明了函数期望的参数个数。例如：

```
function doAdd(iNum) {
  alert(iNum + 10);
}

function sayHi() {
  alert("Hi");
}

alert(doAdd.length);    //输出 "1"
alert(sayHi.length);    //输出 "0"
```

函数 doAdd() 定义了一个参数，因此它的 length 是 1；sayHi() 没有定义参数，所以 length 是 0。

记住，无论定义了几个参数，ECMAScript 可以接受任意多个参数（最多 25 个），这一点在《函数概述》这一章中讲解过。属性 length 只是为查看默认情况下预期的参数个数提供了一种简便方式。

Function 对象的方法

Function 对象也有与所有对象共享的 valueOf() 方法和 toString() 方法。这两个方法返回的都是函数的源代码，在调试时尤其有用。例如：

```
function doAdd(iNum) {
  alert(iNum + 10);
}

document.write(doAdd.toString());
```

上面这段代码输出了 doAdd() 函数的文本。！

ECMAScript 闭包（closure）

ECMAScript 最易让人误解的一点是，它支持闭包（**closure**）。

闭包，指的是词法表示包括不被计算的变量的函数，也就是说，函数可以使用函数之外定义的变量。

简单的闭包实例

在 ECMAScript 中使用全局变量是一个简单的闭包实例。请思考下面这段代码：

```
var sMessage = "hello world";

function sayHelloWorld() {
  alert(sMessage);
}

sayHelloWorld();
```

在上面这段代码中，脚本被载入内存后，并没有为函数 sayHelloWorld() 计算变量 sMessage 的值。该函数捕获 sMessage 的值只是为了以后的使用，也就是说，解释程序知道在调用该函数时要检查 sMessage 的值。sMessage 将在函数调用 sayHelloWorld() 时（最后一行）被赋值，显示消息 "hello world"。

复杂的闭包实例

在一个函数中定义另一个会使闭包变得更加复杂。例如：

```
var iBaseNum = 10;

function addNum(iNum1, iNum2) {
  function doAdd() {
    return iNum1 + iNum2 + iBaseNum;
  }
  return doAdd();
}
```

这里，函数 addNum() 包括函数 doAdd() （闭包）。内部函数是一个闭包，因为它将获取外部函数的参数 iNum1 和 iNum2 以及全局变量 iBaseNum 的值。 addNum() 的最后一步调用了 doAdd()，把两个参数和全局变量相加，并返回它们的和。

这里要掌握的重要概念是，doAdd() 函数根本不接受参数，它使用的值是从执行环境中获取的。

可以看到，闭包是 ECMAScript 中非常强大多用的一部分，可用于执行复杂的计算。

提示：就像使用任何高级函数一样，使用闭包要小心，因为它们可能会变得非常复杂。

ECMAScript 对象

ECMAScript 面向对象技术

面向对象术语

对象

ECMA-262 把对象（object）定义为“属性的无序集合，每个属性存放一个原始值、对象或函数”。严格来说，这意味着对象是无特定顺序的值的数组。

尽管 ECMAScript 如此定义对象，但它更通用的定义是基于代码的名词（人、地点或事物）的表示。

类

每个对象都由类定义，可以把类看做对象的配方。类不仅要定义对象的接口（interface）（开发者访问的属性和方法），还要定义对象的内部工作（使属性和方法发挥作用的代码）。编译器和解释程序都根据类的说明构建对象。

实例

程序使用类创建对象时，生成的对象叫作类的实例（instance）。对类生成的对象的个数的唯一限制来自于运行代码的机器的物理内存。每个实例的行为相同，但实例处理一组独立的数据。由类创建对象实例的过程叫做实例化（instantiation）。

在前面的章节我们提到过，ECMAScript 并没有正式的类。相反，ECMA-262 把对象定义描述为对象的配方。这是 ECMAScript 逻辑上的一种折中方案，因为对象定义实际上是对象自身。即使类并不真正存在，我们也把对象定义叫做类，因为大多数开发者对此术语更熟悉，而且从功能上说，两者是等价的。

面向对象语言的要求

一种面向对象语言需要向开发者提供四种基本能力：

1. 封装 - 把相关的信息（无论数据或方法）存储在对象中的能力
2. 聚集 - 把一个对象存储在另一个对象内的能力
3. 继承 - 由另一个类（或多个类）得来类的属性和方法的能力
4. 多态 - 编写能以多种方法运行的函数或方法的能力

ECMAScript 支持这些要求，因此可被是看做面向对象的。

对象的构成

在 ECMAScript 中，对象由特性（attribute）构成，特性可以是原始值，也可以是引用值。如果特性存放的是函数，它将被看作对象的方法（method），否则该特性被看作对象的属性（property）。

ECMAScript 对象应用

对象的创建和销毁都在 **JavaScript** 执行过程中发生，理解这种范式的含义对理解整个语言至关重要。

声明和实例化

对象的创建方式是用关键字 new 后面跟上实例化的类的名字：

```
var oObject = new Object();
var oStringObject = new String();
```

第一行代码创建了 Object 类的一个实例，并把它存储到变量 oObject 中。第二行代码创建了 String 类的一个实例，把它存储在变量 oStringObject 中。如果构造函数无参数，括号则不是必需的。因此可以采用下面的形式重写上面的两行代码：

```
var oObject = new Object;
var oStringObject = new String;
```

对象引用

在前面的章节中，我们介绍了引用类型的概念。在 ECMAScript 中，不能访问对象的物理表示，只能访问对象的引用。每次创建对象，存储在变量中的都是该对象的引用，而不是对象本身。

对象废除

ECMAScript 拥有无用存储单元收集程序（garbage collection routine），意味着不必专门销毁对象来释放内存。当再没有对对象的引用时，称该对象被废除（dereference）了。运行无用存储单元收集程序时，所有废除的对象都被销毁。每当函数执行完它的代码，无用存储单元收集程序都会运行，释放所有的局部变量，还有在一些其他不可预知的情况下，无用存储单元收集程序也会运行。

把对象的所有引用都设置为 null，可以强制性地废除对象。例如：

```
var oObject = new Object;
// do something with the object here
oObject = null;
```

当变量 oObject 设置为 null 后，对第一个创建的对象的引用就不存在了。这意味着下次运行无用存储单元收集程序时，该对象将被销毁。

每用完一个对象后，就将其废除，来释放内存，这是个好习惯。这样还确保不再使用已经不能访问的对象，从而防止程序设计错误的出现。此外，旧的浏览器（如 IE/MAC）没有全面的无用存储单元收集程序，所以在卸载页面时，对象可能不能被正确销毁。废除对象和它的所有特性是确保内存使用正确的最好方法。

注意：废除对象的所有引用时要当心。如果一个对象有两个或更多引用，则要正确废除该对象，必须将其所有引用都设置为 null。

早绑定和晚绑定

所谓绑定（binding），即把对象的接口与对象实例结合在一起的方法。

早绑定（early binding）是指在实例化对象之前定义它的属性和方法，这样编译器或解释程序就能够提前转换机器代码。在 Java 和 Visual Basic 这样的语言中，有了早绑定，就可以在开发环境中使用 IntelliSense（即给开发者提供对象中属性和方法列表的功能）。ECMAScript 不是强类型语言，所以不支持早绑定。

另一方面，晚绑定（late binding）指的是编译器或解释程序在运行前，不知道对象的类型。使用晚绑定，无需检查对象的类型，只需检查对象是否支持属性和方法即可。ECMAScript 中的所有变量都采用晚绑定方法。这样就允许执行大量的对象操作，而无任何惩罚。

ECMAScript 对象类型

在 **ECMAScript** 中，所有对象并非同等创建的。

一般来说，可以创建并使用的对象有三种：本地对象、内置对象和宿主对象。

本地对象

ECMA-262 把本地对象（native object）定义为“独立于宿主环境的 ECMAScript 实现提供的对象”。简单来说，本地对象就是 ECMA-262 定义的类（引用类型）。它们包括：

- Object
- Function
- Array
- String
- Boolean
- Number
- Date
- RegExp
- Error
- EvalError
- RangeError
- ReferenceError
- SyntaxError
- TypeError
- URIError

相关页面

JavaScript 高级教程：ECMAScript 引用类型

JavaScript 高级教程：ECMAScript Function 类

JavaScript 参考手册：Array 对象

JavaScript 参考手册：Boolean 对象

JavaScript 参考手册：Date 对象

JavaScript 参考手册：Number 对象

JavaScript 参考手册：String 对象

JavaScript 参考手册：RegExp 对象

内置对象

ECMA-262 把内置对象（built-in object）定义为“由 ECMAScript 实现提供的、独立于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现”。这意味着开发者不必明确实例化内置对象，它已被实例化了。ECMA-262 只定义了两个内置对象，即 Global 和 Math （它们也是本地对象，根据定义，每个内置对象都是本地对象）。

相关页面

JavaScript 参考手册：Global 对象

JavaScript 参考手册：Math 对象

宿主对象

所有非本地对象都是宿主对象（host object），即由 ECMAScript 实现的宿主环境提供的对象。

所有 BOM 和 DOM 对象都是宿主对象。

相关页面

JavaScript 高级教程：JavaScript 实现

W3School 参考手册：JavaScript 参考手册

W3School 教程：HTML DOM 教程

ECMAScript 对象作用域

作用域指的是变量的适用范围。

公用、私有和受保护作用域

概念

在传统的面向对象程序设计中，主要关注于公用和私有作用域。公用作用域中的对象属性可以从对象外部访问，即开发者创建对象的实例后，就可使用它的公用属性。而私有作用域中的属性只能在对象内部访问，即对于外部世界来说，这些属性并不存在。这意味着如果类定义了私有属性和方法，则它的子类也不能访问这些属性和方法。

受保护作用域也是用于定义私有的属性和方法，只是这些属性和方法还能被其子类访问。

ECMAScript 只有公用作用域

对 ECMAScript 讨论上面这些作用域几乎毫无意义，因为 ECMAScript 中只存在一种作用域 - 公用作用域。ECMAScript 中的所有对象的所有属性和方法都是公用的。因此，定义自己的类和对象时，必须格外小心。记住，所有属性和方法默认都是公用的！

建议性的解决方法

许多开发者都在网上提出了有效的属性作用域模式，解决了 ECMAScript 的这种问题。

由于缺少私有作用域，开发者确定了一个规约，说明哪些属性和方法应该被看做私有的。这种规约规定在属性前后加下划线：

```
obj._color_ = "blue";
```

这段代码中，属性 color 是私有的。注意，下划线并不改变属性是公用属性的事实，它只是告诉其他开发者，应该把该属性看作私有的。

有些开发者还喜欢用单下划线说明私有成员，例如：obj._color。

静态作用域

静态作用域定义的属性和方法任何时候都能从同一位置访问。在 Java 中，类可具有属性和方法，无需实例化该类的对象，即可访问这些属性和方法，例如 java.net.URLEncoder 类，它的函数 encode() 就是静态方法。

ECMAScript 没有静态作用域

严格来说，ECMAScript 并没有静态作用域。不过，它可以给构造函数提供属性和方法。还记得吗，构造函数只是函数。函数是对象，对象可以有属性和方法。例如：

```
function sayHello() {
  alert("hello");
}

sayHello.alternate = function() {
  alert("hi");
}

sayHello();         //输出 "hello"
sayHello.alternate();    //输出 "hi"
```

TIY

这里，方法 alternate() 实际上是函数 sayHello 的方法。可以像调用常规函数一样调用 sayHello() 输出 "hello"，也可以调用 sayHello.alternate() 输出 "hi"。即使如此，alternate() 也是 sayHello() 公用作用域中的方法，而不是静态方法。

关键字 this

this 的功能

在 ECMAScript 中，要掌握的最重要的概念之一是关键字 this 的用法，它用在对象的方法中。关键字 this 总是指向调用该方法的对象，例如：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function() {
  alert(`this.color`);
};

oCar.showColor();        //输出 "red"
```

TIY

在上面的代码中，关键字 this 用在对象的 showColor() 方法中。在此环境中，this 等于 oCar。下面的代码与上面的代码的功能相同：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function() {
  alert(`oCar.color`);
};

oCar.showColor();        //输出 "red"
```

TIY

使用 this 的原因

为什么使用 this 呢？因为在实例化对象时，总是不能确定开发者会使用什么样的变量名。使用 this，即可在任何多个地方重用同一个函数。请思考下面的例子：

```
function showColor() {
  alert(`this.color`);
};

var oCar1 = new Object;
oCar1.color = "red";
oCar1.showColor = showColor;

var oCar2 = new Object;
oCar2.color = "blue";
oCar2.showColor = showColor;

oCar1.showColor();        //输出 "red"
oCar2.showColor();        //输出 "blue"
```

TIY

在上面的代码中，首先用 this 定义函数 showColor()，然后创建两个对象（oCar1 和 oCar2），一个对象的 color 属性被设置为 "red"，另一个对象的 color 属性被设置为 "blue"。两个对象都被赋予了属性 showColor，指向原始的 showColor () 函数（注意这里不存在命名问题，因为一个是全局函数，而另一个是对象的属性）。调用每个对象的 showColor()，oCar1 输出是 "red"，而 oCar2 的输出是 "blue"。这是因为调用 oCar1.showColor() 时，函数中的 this 关键字等于 oCar1。调用 oCar2.showColor() 时，函数中的 this 关键字等于 oCar2。

注意，引用对象的属性时，必须使用 this 关键字。例如，如果采用下面的代码，showColor() 方法不能运行：

```
function showColor() {
  alert(`color`);
};
```

如果不用对象或 this 关键字引用变量，ECMAScript 就会把它看作局部变量或全局变量。然后该函数将查找名为 color 的局部或全局变量，但是不会找到。结果如何呢？该函数将在警告中显示 "null"。

ECMAScript 定义类或对象

使用预定义对象只是面向对象语言的能力的一部分，它真正强大之处在于能够创建自己专用的类和对象。

ECMAScript 拥有很多创建对象或类的方法。

工厂方式

原始的方式

因为对象的属性可以在对象创建后动态定义，所有许多开发者都在 JavaScript 最初引入时编写类似下面的代码：

```
var oCar = new Object;
oCar.color = "blue";
oCar.doors = 4;
oCar.mpg = 25;
oCar.showColor = function() {
  alert(this.color);
};
```

TIY

在上面的代码中，创建对象 car。然后给它设置几个属性：它的颜色是蓝色，有四个门，每加仑油可以跑 25 英里。最后一个属性实际上是指向函数的指针，意味着该属性是个方法。执行这段代码后，就可以使用对象 car。

不过这里有一个问题，就是可能需要创建多个 car 的实例。

解决方案：工厂方式

要解决该问题，开发者创造了能创建并返回特定类型的对象的工厂函数（factory function）。

例如，函数 createCar() 可用于封装前面列出的创建 car 对象的操作：

```
function createCar() {
  var oTempCar = new Object;
  oTempCar.color = "blue";
  oTempCar.doors = 4;
  oTempCar.mpg = 25;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar();
var oCar2 = createCar();
```

TIY

在这里，第一个例子中的所有代码都包含在 createCar() 函数中。此外，还有一行额外的代码，返回 car 对象（oTempCar）作为函数值。调用此函数，将创建新对象，并赋予它所有必要的属性，复制出一个我们在前面说明过的 car 对象。因此，通过这种方法，我们可以很容易地创建 car 对象的两个版本（oCar1 和 oCar2），它们的属性完全一样。

为函数传递参数

我们还可以修改 createCar() 函数，给它传递各个属性的默认值，而不是简单地赋予属性默认值：

```
function createCar(sColor,iDoors,iMpg) {
  var oTempCar = new Object;
  oTempCar.color = sColor;
  oTempCar.doors = iDoors;
  oTempCar.mpg = iMpg;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar("red",4,23);
var oCar2 = createCar("blue",3,25);

oCar1.showColor();        //输出 "red"
oCar2.showColor();        //输出 "blue"
```

TIY

给 createCar() 函数加上参数，即可为要创建的 car 对象的 color、doors 和 mpg 属性赋值。这使两个对象具有相同的属性，却有不同的属性值。

在工厂函数外定义对象的方法

虽然 ECMAScript 越来越正式化，但创建对象的方法却被置之不理，且其规范化至今还遭人反对。一部分是语义上的原因（它看起来不像使用带有构造函数 new 运算符那么正规），一部分是功能上的原因。功能原因在于用这种方式必须创建对象的方法。前面的例子中，每次

调用函数 createCar()，都要创建新函数 showColor()，意味着每个对象都有自己的 showColor() 版本。而事实上，每个对象都共享同一个函数。

有些开发者在工厂函数外定义对象的方法，然后通过属性指向该方法，从而避免这个问题：

```
`function showColor() {
  alert(this.color);
}`

function createCar(sColor,iDoors,iMpg) {
  var oTempCar = new Object;
  oTempCar.color = sColor;
  oTempCar.doors = iDoors;
  oTempCar.mpg = iMpg;
  `oTempCar.showColor = showColor;`
  return oTempCar;
}

var oCar1 = createCar("red",4,23);
var oCar2 = createCar("blue",3,25);

oCar1.showColor();        //输出 "red"
oCar2.showColor();        //输出 "blue"
```

TIY

在上面这段重写的代码中，在函数 createCar() 之前定义了函数 showColor()。在 createCar() 内部，赋予对象一个指向已经存在的 showColor() 函数的指针。从功能上讲，这样解决了重复创建函数对象的问题；但是从语义上讲，该函数不太像是对象的方法。

所有这些问题都引发了开发者定义的构造函数的出现。

构造函数方式

创建构造函数就像创建工厂函数一样容易。第一步选择类名，即构造函数的名字。根据惯例，这个名字的首字母大写，以使它与首字母通常是小写的变量名分开。除了这点不同，构造函数看起来很像工厂函数。请考虑下面的例子：

```
function Car(sColor,iDoors,iMpg) {
  `this`.color = sColor;
  `this`.doors = iDoors;
  `this`.mpg = iMpg;
  `this`.showColor = function() {
    alert(this.color);
  };
}

var oCar1 = `new` Car("red",4,23);
var oCar2 = `new` Car("blue",3,25);
```

TIY

下面为您解释上面的代码与工厂方式的差别。首先在构造函数内没有创建对象，而是使用 this 关键字。使用 new 运算符构造函数时，在执行第一行代码前先创建一个对象，只有用 this 才能访问该对象。然后可以直接赋予 this 属性，默认情况下是构造函数的返回值（不必明确使用 return 运算符）。

现在，用 new 运算符和类名 Car 创建对象，就更像 ECMAScript 中一般对象的创建方式了。

你也许会问，这种方式在管理函数方面是否存在于前一种方式相同的问题呢？是的。

就像工厂函数，构造函数会重复生成函数，为每个对象都创建独立的函数版本。不过，与工厂函数相似，也可以用外部函数重写构造函数，同样地，这么做语义上无任何意义。这正是下面要讲的原型方式的优势所在。

原型方式

该方式利用了对象的 prototype 属性，可以把它看成创建新对象所依赖的原型。

这里，首先用空构造函数来设置类名。然后所有的属性和方法都被直接赋予 prototype 属性。我们重写了前面的例子，代码如下：

```
function Car() {
}

Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
Car.prototype.showColor = function() {
  alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();
```

TIY

在这段代码中，首先定义构造函数（Car），其中无任何代码。接下来的几行代码，通过给 Car 的 prototype 属性添加属性去定义 Car 对象的属性。调用 new Car() 时，原型的所有属性都被立即赋予要创建的对象，意味着所有 Car 实例存放的都是指向 showColor() 函数的指针。从语义上讲，所有属性看起来都属于一个对象，因此解决了前面两种方式存在的问题。

此外，使用这种方式，还能用 instanceof 运算符检查给定变量指向的对象的类型。因此，下面的代码将输出 TRUE：

```
alert(oCar1 instanceof Car);    //输出 "true"
```

原型方式的问题

原型方式看起来是个不错的解决方案。遗憾的是，它并不尽如人意。

首先，这个构造函数没有参数。使用原型方式，不能通过给构造函数传递参数来初始化属性的值，因为 Car1 和 Car2 的 color 属性都等于 "blue"，doors 属性都等于 4，mpg 属性都等于 25。这意味着必须在对象创建后才能改变属性的默认值，这点很令人讨厌，但还没完。真正的问题出现在属性指向的是对象，而不是函数时。函数共享不会造成问题，但对象却很少被多个实例共享。请思考下面的例子：

```
function Car() {
}

Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
`Car.prototype.drivers = new Array("Mike","John");`
Car.prototype.showColor = function() {
  alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();

`oCar1.drivers.push("Bill");`

alert(oCar1.drivers);    //输出 "Mike,John,Bill"
alert(oCar2.drivers);    //输出 "Mike,John,Bill"
```

TIY

上面的代码中，属性 drivers 是指向 Array 对象的指针，该数组中包含两个名字 "Mike" 和 "John"。由于 drivers 是引用值，Car 的两个实例都指向同一个数组。这意味着给 oCar1.drivers 添加值 "Bill"，在 oCar2.drivers 中也能看到。输出这两个指针中的任何一个，结果都是显示字符串 "Mike,John,Bill"。

由于创建对象时有这么多问题，你一定会想，是否有种合理的创建对象的方法呢？答案是有，需要联合使用构造函数和原型方式。

混合的构造函数/原型方式

联合使用构造函数和原型方式，就可像用其他程序设计语言一样创建对象。这种概念非常简单，即用构造函数定义对象的所有非函数属性，用原型方式定义对象的函数属性（方法）。结果是，所有函数都只创建一次，而每个对象都具有自己的对象属性实例。

我们重写了前面的例子，代码如下：

```
function Car(sColor,iDoors,iMpg) {
  this.color = sColor;
  this.doors = iDoors;
  this.mpg = iMpg;
  this.drivers = new Array("Mike","John");
}

Car.prototype.showColor = function() {
  alert(this.color);
};

var oCar1 = new Car("red",4,23);
var oCar2 = new Car("blue",3,25);

oCar1.drivers.push("Bill");

alert(oCar1.drivers);    //输出 "Mike,John,Bill"
alert(oCar2.drivers);    //输出 "Mike,John"
```

TIY

现在就更像创建一般对象了。所有的非函数属性都在构造函数中创建，意味着又能够用构造函数的参数赋予属性默认值了。因为只创建 showColor() 函数的一个实例，所以没有内存浪费。此外，给 oCar1 的 drivers 数组添加 "Bill" 值，不会影响到 oCar2 的数组，所以输出这些数组的值时，oCar1.drivers 显示的是 "Mike,John,Bill"，而 oCar2.drivers 显示的是 "Mike,John"。因为使用了原型方式，所以仍然能利用 instanceof 运算符来判断对象的类型。

这种方式是 ECMAScript 采用的主要方式，它具有其他方式的特性，却没有他们的副作用。不过，有些开发者仍觉得这种方法不够完美。

动态原型方法

对于习惯使用其他语言的开发者来说，使用混合的构造函数/原型方式感觉不那么和谐。毕竟，定义类时，大多数面向对象语言都对属性和方法进行了视觉上的封装。请考虑下面的 Java 类：

```
class Car {
  public String color = "blue";
  public int doors = 4;
  public int mpg = 25;

  public Car(String color, int doors, int mpg) {
    this.color = color;
    this.doors = doors;
    this.mpg = mpg;
  }

  public void showColor() {
    System.out.println(color);
  }
}
```

Java 很好地打包了 Car 类的所有属性和方法，因此看见这段代码就知道它要实现什么功能，它定义了一个对象的信息。批评混合的构造函数/原型方式的人认为，在构造函数内部找属性，在其外部找方法的做法不合逻辑。因此，他们设计了动态原型方法，以提供更友好的编码风格。

动态原型方法的基本想法与混合的构造函数/原型方式相同，即在构造函数内定义非函数属性，而函数属性则利用原型属性定义。唯一的区别是赋予对象方法的位置。下面是用动态原型方法重写的 Car 类：

```
function Car(sColor,iDoors,iMpg) {
  this.color = sColor;
  this.doors = iDoors;
  this.mpg = iMpg;
  this.drivers = new Array("Mike","John");

  if (`typeof Car._initialized == "undefined"`) {
    Car.prototype.showColor = function() {
      alert(this.color);
    };

    `Car._initialized = true;`
  }
}
```

TIY

直到检查 typeof Car._initialized 是否等于 "undefined" 之前，这个构造函数都未发生变化。这行代码是动态原型方法中最重要的部分。如果这个值未定义，构造函数将用原型方式继续定义对象的方法，然后把 Car._initialized 设置为 true。如果这个值定义了（它的值为 true 时，typeof 的值为 Boolean），那么就不再创建该方法。简而言之，该方法使用标志（_initialized）来判断是否已给原型赋予了任何方法。该方法只创建并赋值一次，传统的 OOP 开发者会高兴地发现，这段代码看起来更像其他语言中的类定义了。

混合工厂方式

这种方式通常是在不能应用前一种方式时的变通方法。它的目的是创建假构造函数，只返回另一种对象的新实例。

这段代码看起来与工厂函数非常相似：

```
function Car() {
  `var oTempCar = new Object;`
  oTempCar.color = "blue";
  oTempCar.doors = 4;
  oTempCar.mpg = 25;
  oTempCar.showColor = function() {
    alert(this.color);
  };

  return oTempCar;
}
```

TIY

与经典方式不同，这种方式使用 new 运算符，使它看起来像真正的构造函数：

```
var car = new Car();
```

由于在 Car() 构造函数内部调用了 new 运算符，所以将忽略第二个 new 运算符（位于构造函数之外），在构造函数内部创建的对象被传递回变量 car。

这种方式在对象方法的内部管理方面与经典方式有着相同的问题。强烈建议：除非万不得已，还是避免使用这种方式。

采用哪种方式

如前所述，目前使用最广泛的是混合的构造函数/原型方式。此外，动态原始方法也很流行，在功能上与构造函数/原型方式等价。可以采用这两种方式中的任何一种。不过不要单独使用经典的构造函数或原型方式，因为这样会给代码引入问题。

实例

对象令人感兴趣的一点是用它们解决问题的方式。ECMAScript 中最常见的一个问题是字符串连接的性能。与其他语言类似，ECMAScript 的字符串是不可变的，即它们的值不能改变。请考虑下面的代码：

```
var str = "hello ";
str += "world";
```

实际上，这段代码在幕后执行的步骤如下：

1. 创建存储 "hello " 的字符串。
2. 创建存储 "world" 的字符串。
3. 创建存储连接结果的字符串。
4. 把 str 的当前内容复制到结果中。
5. 把 "world" 复制到结果中。
6. 更新 str，使它指向结果。

每次完成字符串连接都会执行步骤 2 到 6，使得这种操作非常消耗资源。如果重复这一过程几百次，甚至几千次，就会造成性能问题。解决方法是用 Array 对象存储字符串，然后用 join() 方法（参数是空字符串）创建最后的字符串。想象用下面的代码代替前面的代码：

```
var arr = new Array();
arr[0] = "hello ";
arr[1] = "world";
var str = arr.join("");
```

这样，无论数组中引入多少字符串都不成问题，因为只在调用 join() 方法时才会发生连接操作。此时，执行的步骤如下：

1. 创建存储结果的字符串
2. 把每个字符串复制到结果中的合适位置

虽然这种解决方案很好，但还有更好的方法。问题是，这段代码不能确切反映出它的意图。要使它更容易理解，可以用 StringBuffer 类打包该功能：

```
function StringBuffer () {
  this._strings_ = new Array();
}

StringBuffer.prototype.append = function(str) {
  this._strings_.push(str);
};

StringBuffer.prototype.toString = function() {
  return this._strings_.join("");
};
```

这段代码首先要注意的是 strings 属性，本意是私有属性。它只有两个方法，即 append() 和 toString() 方法。append() 方法有一个参数，它把该参数附加到字符串数组中，toString() 方法调用数组的 join 方法，返回真正连接成的字符串。要用 StringBuffer 对象连接一组字符串，可以用下面的代码：

```
var buffer = new StringBuffer ();
buffer.append("hello ");
buffer.append("world");
var result = buffer.toString();
```

TIY

可用下面的代码测试 StringBuffer 对象和传统的字符串连接方法的性能：

```
var d1 = new Date();
var str = "";
for (var i=0; i < 10000; i++) {
    str += "text";
}
var d2 = new Date();

document.write("Concatenation with plus: "
 + (d2.getTime() - d1.getTime()) + " milliseconds");

var buffer = new StringBuffer();
d1 = new Date();
for (var i=0; i < 10000; i++) {
    buffer.append("text");
}
var result = buffer.toString();
d2 = new Date();

document.write("<br />Concatenation with StringBuffer: "
 + (d2.getTime() - d1.getTime()) + " milliseconds");
```

TIY

这段代码对字符串连接进行两个测试，第一个使用加号，第二个使用 StringBuffer 类。每个操作都连接 10000 个字符串。日期值 d1 和 d2 用于判断完成操作需要的时间。请注意，创建 Date 对象时，如果没有参数，赋予对象的是当前的日期和时间。要计算连接操作历经多少时间，把日期的毫秒表示（用 getTime() 方法的返回值）相减即可。这是衡量 JavaScript 性能的常见方法。该测试的结果可以帮助您比较使用 StringBuffer 类与使用加号的效率差异。

ECMAScript 修改对象

通过使用 **ECMAScript**，不仅可以创建对象，还可以修改已有对象的行为。

prototype 属性不仅可以定义构造函数的属性和方法，还可以为本地对象添加属性和方法。

创建新方法

通过已有的方法创建新方法

可以用 prototype 属性为任何已有的类定义新方法，就像处理自己的类一样。例如，还记得 Number 类的 toString() 方法吗？如果给它传递参数 16，它将输出十六进制的字符串。如果这个方法的参数是 2，那么它将输出二进制的字符串。我们可以创建一个方法，可以把数字对象直接转换为十六进制字符串。创建这个方法非常简单：

```
Number.prototype.toHexString = function() {
  return this.toString(16);
};
```

在此环境中，关键字 this 指向 Number 的实例，因此可完全访问 Number 的所有方法。有了这段代码，可实现下面的操作：

```
var iNum = 15;
alert(iNum.toHexString());        //输出 "F"
```

TIY

由于数字 15 等于十六进制中的 F，因此警告将显示 "F"。

重命名已有方法

我们还可以为已有的方法命名更易懂的名称。例如，可以给 Array 类添加两个方法 enqueue() 和 dequeue()，只让它们反复调用已有的 push() 和 shift() 方法即可：

```
Array.prototype.enqueue = function(vItem) {
  this.push(vItem);
};

Array.prototype.dequeue = function() {
  return this.shift();
};
```

TIY

添加与已有方法无关的方法

当然，还可以添加与已有方法无关的方法。例如，假设要判断某个项在数组中的位置，没有本地方法可以做这种事情。我们可以轻松地创建下面的方法：

```
Array.prototype.indexOf = function (vItem) {
  for (var i=0; i<this.length; i++) {
    if (vItem == this[i]) {
      return i;
    }
  }

  return -1;
}
```

该方法 indexOf() 与 String 类的同名方法保持一致，在数组中检索每个项，直到发现与传进来的项相同的项目为止。如果找到相同的项，则返回该项的位置，否则，返回 -1。有了这种定义，我们可以编写下面的代码：

```
var aColors = new Array("red","green","blue");
alert(aColors.indexOf("green"));    //输出 "1"
```

TIY

为本地对象添加新方法

最后，如果想给 ECMAScript 中每个本地对象添加新方法，必须在 Object 对象的 prototype 属性上定义它。前面的章节我们讲过，所有本地对象都继承了 Object 对象，所以对 Object 对象做任何改变，都会反应在所有本地对象上。例如，如果想添加一个用警告输出对象的当前值的方法，可以采用下面的代码：

```
Object.prototype.showValue = function () {
  alert(this.valueOf());
};

var str = "hello";
var iNum = 25;
str.showValue();        //输出 "hello"
iNum.showValue();        //输出 "25"
```

TIY

这里，String 和 Number 对象都从 Object 对象继承了 showValue() 方法，分别在它们的对象上调用该方法，将显示 "hello" 和 "25"。

重定义已有方法

就像能给已有的类定义新方法一样，也可重定义已有的方法。如前面的章节所述，函数名只是指向函数的指针，因此可以轻松地指向其他函数。如果修改了本地方法，如 toString()，会出现什么情况呢？

```
Function.prototype.toString = function() {
  return "Function code hidden";
}
```

前面的代码完全合法，运行结果完全符合预期：

```
function sayHi() {
  alert("hi");
}

alert(sayHi.toString());	//输出 "Function code hidden"
```

TIY

也许你还记得，Function 对象这一章中介绍过 Function 的 toString() 方法通常输出的是函数的源代码。覆盖该方法，可以返回另一个字符串（在这个例子中，可以返回 "Function code hidden"）。不过，toString() 指向的原始函数怎么了呢？它将被无用存储单元回收程序回收，因为它被完全废弃了。没有能够恢复原始函数的方法，所以在覆盖原始方法前，比较安全的做法是存储它的指针，以便以后的使用。有时你甚至可能在新方法中调用原始方法：

```
`Function.prototype.originalToString = Function.prototype.toString;`

Function.prototype.toString = function() {
  if (this.originalToString().length > 100) {
    return "Function too long to display.";
  } else {
    return this.originalToString();
  }
};
```

TIY

在这段代码中，第一行代码把对当前 toString() 方法的引用保存在属性 originalToString 中。然后用定制的方法覆盖了 toString() 方法。新方法将检查该函数源代码的长度是否大于 100。如果是，就返回错误信息，说明该函数代码太长，否则调用 originalToString() 方法，返回函数的源代码。

极晚绑定（Very Late Binding）

从技术上讲，根本不存在极晚绑定。本书采用该术语描述 ECMAScript 中的一种现象，即能够在对象实例化后再定义它的方法。例如：

```
var o = new Object();

Object.prototype.sayHi = function () {
  alert("hi");
};

o.sayHi();
```

TIY

在大多数程序设计语言中，必须在实例化对象之前定义对象的方法。这里，方法 sayHi() 是在创建 Object 类的一个实例之后来添加进来的。在传统语言中不仅没听说过这种操作，也没听说过该方法还会自动赋予 Object 对象的实例并能立即使用（接下来的一行）。

注意：不建议使用极晚绑定方法，因为很难对其跟踪和记录。不过，还是应该了解这种可能。

ECMAScript 继承

ECMAScript 继承机制实例

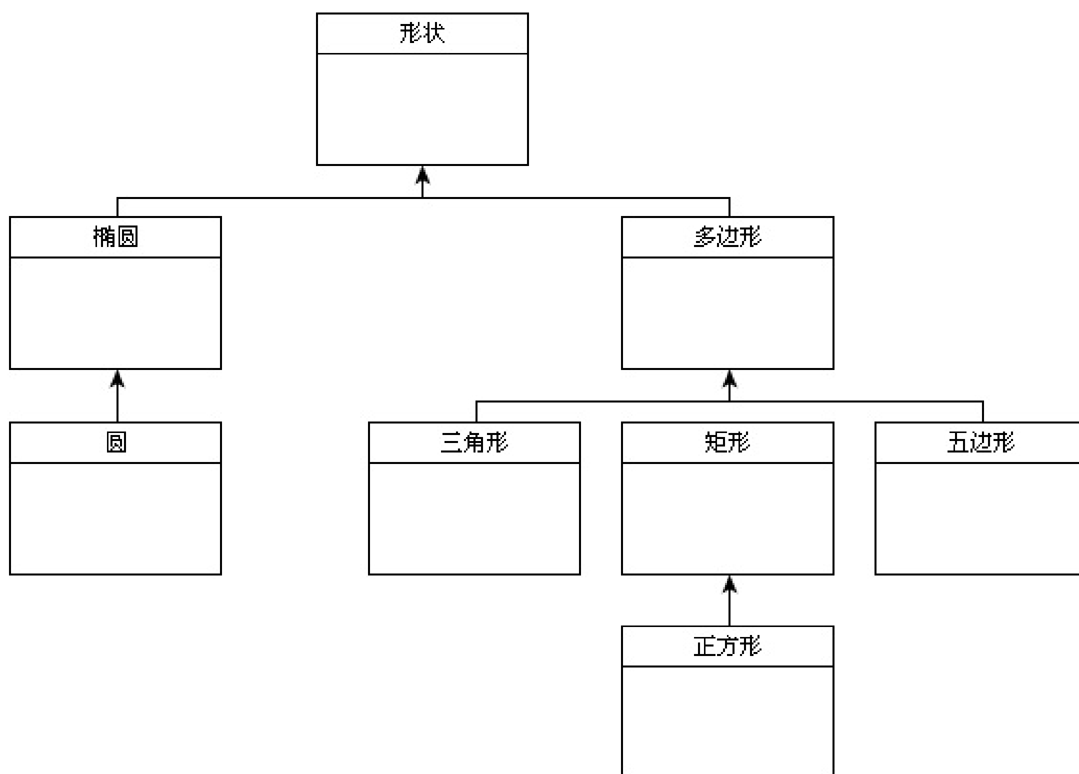
本节使用一个经典的例子解释 **ECMAScript** 的继承机制。

继承机制实例

说明继承机制最简单的方式是，利用一个经典的例子 - 几何形状。实际上，几何形状只有两种，即椭圆形（是圆形的）和多边形（具有一定数量的边）。圆是椭圆的一种，它只有一个焦点。三角形、矩形和五边形都是多边形的一种，具有不同数量的边。正方形是矩形的一种，所有的边等长。这就构成了一种完美的继承关系。

在这个例子中，形状（Shape）是椭圆形（Ellipse）和多边形（Polygon）的基类（base class）（所有类都由它继承而来）。椭圆具有一个属性 *foci*，说明椭圆具有的焦点的个数。圆形（Circle）继承了椭圆形，因此圆形是椭圆形的子类（subclass），椭圆形是圆形的超类（superclass）。同样，三角形（Triangle）、矩形（Rectangle）和五边形（Pentagon）都是多边形的子类，多边形是它们的超类。最后，正方形（Square）继承了矩形。

最好用图来解释这种继承关系，这是 UML（统一建模语言）的用武之地。UML 的主要用途之一是，可视化地表示像继承这样的复杂对象关系。下面的图示是解释 Shape 和它的子类之间关系的 UML 图示：



在 UML 中，每个方框表示一个类，由类名说明。三角形 、矩形和五边形顶部的线段汇集在一起，指向形状，说明这些类都由形状继承而来。同样，从正方形指向矩形的箭头说明了它们之间的继承关系。

ECMAScript 继承机制实现

继承机制的实现

要用 ECMAScript 实现继承机制，您可以从要继承的基类入手。所有开发者定义的类都可作为基类。出于安全原因，本地类和宿主类不能作为基类，这样可以防止公用访问编译过的浏览器级的代码，因为这些代码可以被用于恶意攻击。

选定基类后，就可以创建它的子类了。是否使用基类完全由你决定。有时，你可能想创建一个不能直接使用的基类，它只是用于给子类提供通用的函数。在这种情况下，基类被看作抽象类。

尽管 ECMAScript 并没有像其他语言那样严格地定义抽象类，但有时它的确会创建一些不允许使用的类。通常，我们称这种类为抽象类。

创建的子类将继承超类的所有属性和方法，包括构造函数及方法的实现。记住，所有属性和方法都是公用的，因此子类可直接访问这些方法。子类还可添加超类中没有的新属性和方法，也可以覆盖超类的属性和方法。

继承的方式

和其他功能一样，ECMAScript 实现继承的方式不止一种。这是因为 JavaScript 中的继承机制并不是明确规定的，而是通过模仿实现的。这意味着所有的继承细节并非完全由解释程序处理。作为开发者，你有权决定最适用的继承方式。

下面为您介绍几种具体的继承方式。

对象冒充

构想原始的 ECMAScript 时，根本没打算设计对象冒充（object masquerading）。它是在开发者开始理解函数的工作方式，尤其是如何在函数环境中使用 this 关键字后才发展出来。

其原理如下：构造函数使用 this 关键字给所有属性和方法赋值（即采用类声明的构造函数方式）。因为构造函数只是一个函数，所以可使 ClassA 构造函数成为 ClassB 的方法，然后调用它。ClassB 就会收到 ClassA 的构造函数中定义的属性和方法。例如，用下面的方式定义 ClassA 和 ClassB：

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () {
        alert(this.color);
    };
}

function ClassB(sColor) {
}
```

还记得吗？关键字 this 引用的是构造函数当前创建的对象。不过在这个方法中，this 指向的所属的对象。这个原理是把 ClassA 作为常规函数来建立继承机制，而不是作为构造函数。如下使用构造函数 ClassB 可以实现继承机制：

```
function ClassB(sColor) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;
}
```

在这段代码中，为 ClassA 赋予了方法 newMethod（请记住，函数名只是指向它的指针）。然后调用该方法，传递给它的是 ClassB 构造函数的参数 sColor。最后一行代码删除了对 ClassA 的引用，这样以后就不能再调用它。

所有新属性和新方法都必须在删除了新方法的代码行后定义。否则，可能会覆盖超类的相关属性和方法：

```
function ClassB(sColor, sName) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;

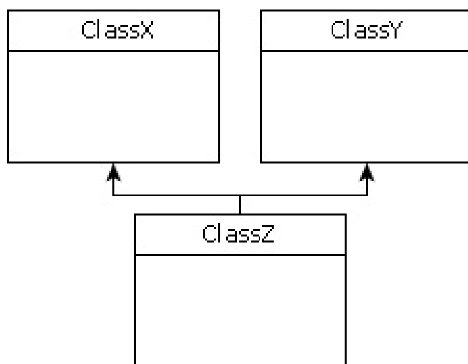
    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

为证明前面的代码有效，可以运行下面的例子：

```
var objA = new ClassA("blue");
var objB = new ClassB("red", "John");
objA.sayColor();    //输出 "blue"
objB.sayColor();    //输出 "red"
objB.sayName();        //输出 "John"
```

对象冒充可以实现多重继承

有趣的是，对象冒充可以支持多重继承。也就是说，一个类可以继承多个超类。用 UML 表示的多重继承机制如下图所示：



例如，如果存在两个类 ClassX 和 ClassY，ClassZ 想继承这两个类，可以使用下面的代码：

```
function ClassZ() {
    this.newMethod = ClassX;
    this.newMethod();
    delete this.newMethod;

    this.newMethod = ClassY;
    this.newMethod();
    delete this.newMethod;
}
```

TIY

这里存在一个弊端，如果存在两个类 ClassX 和 ClassY 具有同名的属性或方法，ClassY 具有高优先级。因为它从后面的类继承。除这点小问题之外，用对象冒充实现多重继承机制轻而易举。

由于这种继承方法的流行，ECMAScript 的第三版为 Function 对象加入了两个方法，即 call() 和 apply()。

call() 方法

call() 方法是与经典的对象冒充方法最相似的方法。它的第一个参数用作 this 的对象。其他参数都直接传递给函数自身。例如：

```
function sayColor(sPrefix,sSuffix) {
    alert(sPrefix + this.color + sSuffix);
};

var obj = new Object();
obj.color = "blue";

sayColor.call(obj, "The color is ", "a very nice color indeed.");
```

在这个例子中，函数 sayColor() 在对象外定义，即使它不属于任何对象，也可以引用关键字 this。对象 obj 的 color 属性等于 blue。调用 call() 方法时，第一个参数是 obj，说明应该赋予 sayColor() 函数中的 this 关键字值是 obj。第二个和第三个参数是字符串。它们与 sayColor()

函数中的参数 sPrefix 和 sSuffix 匹配，最后生成的消息 "The color is blue, a very nice color indeed." 将被显示出来。

要与继承机制的对象冒充方法一起使用该方法，只需将前三行的赋值、调用和删除代码替换即可：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(color);
    //delete this.newMethod;
    ClassA.call(this, sColor);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

TIY

这里，我们需要让 ClassA 中的关键字 this 等于新创建的 ClassB 对象，因此 this 是第一个参数。第二个参数 sColor 对两个类来说都是唯一的参数。

apply() 方法

apply() 方法有两个参数，用作 this 的对象和要传递给函数的参数的数组。例如：

```
function sayColor(sPrefix,sSuffix) {
    alert(sPrefix + this.color + sSuffix);
};

var obj = new Object();
obj.color = "blue";

sayColor.apply(obj, new Array("The color is ", "a very nice color indeed."));
```

这个例子与前面的例子相同，只是现在调用的是 apply() 方法。调用 apply() 方法时，第一个参数仍是 obj，说明应该赋予 sayColor() 函数中的 this 关键字值是 obj。第二个参数是由两个字符串构成的数组，与 sayColor() 函数中的参数 sPrefix 和 sSuffix 匹配，最后生成的消息仍是 "The color is blue, a very nice color indeed."，将被显示出来。

该方法也用于替换前三行的赋值、调用和删除新方法的代码：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(color);
    //delete this.newMethod;
    ClassA.apply(this, new Array(sColor));

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

同样的，第一个参数仍是 this，第二个参数是只有一个值 color 的数组。可以把 ClassB 的整个 arguments 对象作为第二个参数传递给 apply() 方法：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(color);
    //delete this.newMethod;
    ClassA.apply(this, arguments);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

TIY

当然，只有超类中的参数顺序与子类中的参数顺序完全一致时才可以传递参数对象。如果不是，就必须创建一个单独的数组，按照正确的顺序放置参数。此外，还可使用 call() 方法。

原型链（prototype chaining）

继承这种形式在 ECMAScript 中原本是用于原型链的。上一章介绍了定义类的原型方式。原型链扩展了这种方式，以一种有趣的方式实现继承机制。

在上一章学过，prototype 对象是个模板，要实例化的对象都以这个模板为基础。总而言之，prototype 对象的任何属性和方法都被传递给那个类的所有实例。原型链利用这种功能来实现继承机制。

如果用原型方式重定义前面例子中的类，它们将变为下列形式：

```
function ClassA() {
}

ClassA.prototype.color = "blue";
ClassA.prototype.sayColor = function () {
    alert(this.color);
};

function ClassB() {
}

`ClassB.prototype = new ClassA();`
```

原型方式的神奇之处在于突出显示的蓝色代码行。这里，把 ClassB 的 prototype 属性设置成 ClassA 的实例。这很有意思，因为想要 ClassA 的所有属性和方法，但又不想逐个将它们 ClassB 的 prototype 属性。还有比把 ClassA 的实例赋予 prototype 属性更好的方法吗？

注意：调用 ClassA 的构造函数，没有给它传递参数。这在原型链中是标准做法。要确保构造函数没有任何参数。

与对象冒充相似，子类的所有属性和方法都必须出现在 prototype 属性被赋值后，因为在它之前赋值的所有方法都会被删除。为什么？因为 prototype 属性被替换成了新对象，添加了新方法的原始对象将被销毁。所以，为 ClassB 类添加 name 属性和 sayName() 方法的代码如下：

```
function ClassB() {
}

ClassB.prototype = new ClassA();

ClassB.prototype.name = "";
ClassB.prototype.sayName = function () {
    alert(this.name);
};
```

可通过运行下面的例子测试这段代码：

```
var objA = new ClassA();
var objB = new ClassB();
objA.color = "blue";
objB.color = "red";
objB.name = "John";
objA.sayColor();
objB.sayColor();
objB.sayName();
```

TIY

此外，在原型链中，instanceof 运算符的运行方式也很独特。对 ClassB 的所有实例，instanceof 为 ClassA 和 ClassB 都返回 true。例如：

```
var objB = new ClassB();
alert(objB instanceof ClassA);    //输出 "true"
alert(objB instanceof ClassB);    //输出 "true"
```

在 ECMAScript 的弱类型世界中，这是极其有用的工具，不过使用对象冒充时不能使用它。

原型链的弊端是不支持多重继承。记住，原型链会用另一类型的对象重写类的 prototype 属性。

混合方式

这种继承方式使用构造函数定义类，并非使用任何原型。对象冒充的主要问题是必须使用构造函数方式，这不是最好的选择。不过如果使用原型链，就无法使用带参数的构造函数了。开发者如何选择呢？答案很简单，两者都用。

在前一章，我们曾经讲解过创建类的最好方式是用构造函数定义属性，用原型定义方法。这种方式同样适用于继承机制，用对象冒充继承构造函数的属性，用原型链继承 prototype 对象的方法。用这两种方式重写前面的例子，代码如下：

```
function ClassA(sColor) {
    this.color = sColor;
}

ClassA.prototype.sayColor = function () {
    alert(this.color);
};

function ClassB(sColor, sName) {
    `ClassA.call(this, sColor);`
    this.name = sName;
}

`ClassB.prototype = new ClassA();`

ClassB.prototype.sayName = function () {
    alert(this.name);
};
```

在此例子中，继承机制由两行突出显示的蓝色代码实现。在第一行突出显示的代码中，在 ClassB 构造函数中，用对象冒充继承 ClassA 类的 sColor 属性。在第二行突出显示的代码中，用原型链继承 ClassA 类的方法。由于这种混合方式使用了原型链，所以 instanceof 运算符仍能正确运行。

下面的例子测试了这段代码：

```
var objA = new ClassA("blue");
var objB = new ClassB("red", "John");
objA.sayColor();    //输出 "blue"
objB.sayColor();    //输出 "red"
objB.sayName();    //输出 "John"
```

TIY

Google 地图API

Google 地图API Key

开始学习本教程前，你需要拥有一个免费的 Google 地图 API key。

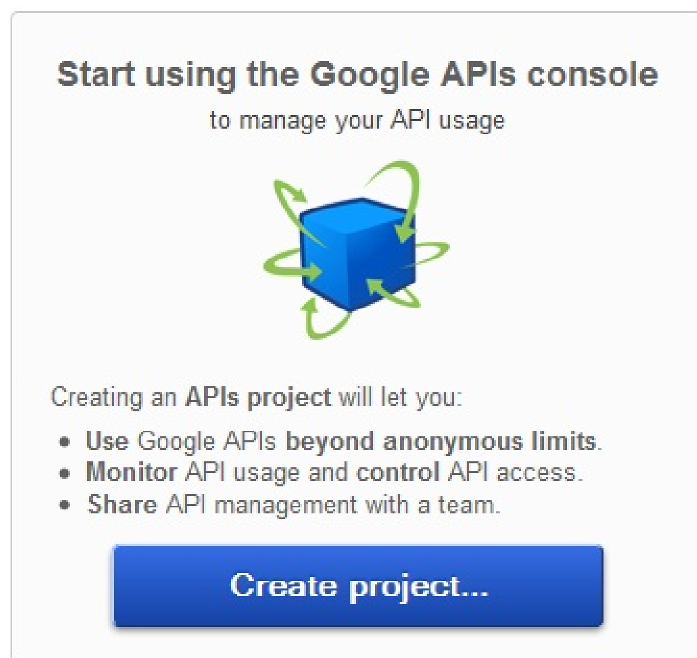
开始学习？

开始学习本教程前，你需要在Google上申请一个指定的API key。

通过以下步骤我们可以免费获取 API key 。

访问 https://code.google.com/apis/console/, 使用你的Google账号登陆。

登陆后会出现如下界面：



点击 "Create Project" 按钮。

在服务列表中找到 **Google Maps API v3,** 然后点击 "off"（关闭） 让其开启该服务器

在下一个步骤中，选择"I Agree..." 然后点击 "Accept" 按钮。 现在你在服务列表中应该就可以看到 Google Maps API v3 已经变为 "on"（开启）状态。

接着在左侧菜单中点击 "API Access" ，在右侧栏中将看到以下提示 "Create an OAuth 2.0 client id..."。

点击 "Create an OAuth 2.0 client id..."，将弹出一个表单，表单需要你填入你的项目名称，项目图片或者logo，然后点击 "Next" 按钮。

然后，我们需要选择应用类型 ("Web application" ：网站应用), 然后填写你的站点地址，之后点击 "Create Client Id" 按钮即可。

最后我们就可以得到我们需要的 API key,如下图所示：

Simple API Access

Use API keys to identify your project when you do not need to access user data. Learn more

Key for browser apps (with referers)

API key:	AIzaSyDY0kkJiTPVd2U7aTOAwhc9ySH6oHxOIYM
Referers:	Any referer allowed
Activated on:	Mar 20, 2012 5:46 AM
Activated by:	support@w3schools.com – you

Create new Server key... | Create new Browser key...



注意： 保存你的API key！ (在填写的指定 URL 中开发所有的 Google 地图应用你需要使用该API key)。

Google Maps 基础

创建一个简单的 Google 地图

现在让我们创建一个简单的 Google 地图。

以下是显示了英国伦敦的 Google 地图：

实例

```
<!DOCTYPE html>
<html>
<head>
<script src="http://maps.googleapis.com/maps/api/js?key=AIzaSyDY0kkJiTPVd2U7aTOAwhc9ySH6o
</script>

<script>
function initialize()
{
var mapProp = {
  center:new google.maps.LatLng(51.508742,-0.120850),
  zoom:5,
  mapTypeId:google.maps.MapTypeId.ROADMAP
  };
var map=new google.maps.Map(document.getElementById("googleMap")
  ,mapProp);
}

google.maps.event.addDomListener(window, 'load', initialize);
</script>
</head>

<body>
<div id="googleMap" style="width:500px;height:380px;"></div>

</body>
</html>
```

实例解析

我们以以上实例来解析 Google 地图的创建过程。

应用为什么要声明 HTML5?

```
<!DOCTYPE html>
```

大多数浏览器使用 "标准模式" 的 HTML5 文档渲染页面，这就意味着你的应用是兼容各大浏览器的。

另外，如果没有DOCTYPE标签，浏览器则使用混杂模式 (quirks mode)进行渲染页面内容。

提示： 应该注意的是一些"混杂模式 "中的CSS并不能使用与标准模式中。在具体的应用中，所有基于百分比的大小都必须从父块元素继承 。如果在父模块中没有指定大小，默认值为 0 x 0 像素。如果你想使用百分比，可以在<style> 标签中声明，如下所示：

```
<style type="text/css">
html {height:100%}
body {height:100%;margin:0;padding:0}
#googleMap {height:100%}
</style>
```

这个样式声明表明地图模块的（GoogleMap）应 HTML高度为100%。

添加 Google 地图 API Key

在以下实例中第一个<script> 标签中必须包含 Google 地图 API：

```
<script src="http://maps.googleapis.com/maps/api/js?key=_**YOUR_API_KEY**_&sensor=**_TRUE
```

将google生成的 API key 放置于 **key** 参数中(key=*YOUR_API_KEY*)。

The **sensor** 参数是必须的，该参数用于指明应用程序是否使用一个传感器 (类似 GPS 导航)来定位用户的位置。参数值可以设置为 true 或者 false。

HTTPS

如果你的应用是安全的HTTP(HTTPS:HTTP Secure)应用,你可以使用 HTTPS 来加载 Google 地图 API：

```
<script src="https://maps.googleapis.com/maps/api/js?key=_**YOUR_API_KEY**_&sensor=**_TRU
```

异步加载

同样我们也可以在页面完全载入后再加载 Google 地图 API。

以下实例使用了 window.onload 来实现页面完全载入后加载 Google 地图 。 loadScript() 函数创建了加载 Google 地图 API <script> 标签。此外在标签的末尾添加了 callback=initialize 参数， initialize()作为回调函数会在API完全载入后执行：

实例

```
function loadScript()
{
var script = document.createElement("script");
script.src = "http://maps.googleapis.com/maps/api/js?      key=AIzaSyDY0kkJiTPVd2U7aTOAwh
}

window.onload = loadScript;
```

定义地图属性

在初始化地图前，我们需要先创建一个 Map 属性对象来定义一些地图的属性：

```
var mapProp = {
  center:new google.maps.LatLng(51.508742,-0.120850),
  zoom:7,
  mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

center（中心点）

中心属性指定了地图的中心，该中心通过坐标（纬度，经度）在地图上创建一个中心点。

Zoom（缩放级数）

zoom 属性指定了地图的 缩放级数。zoom: 0 显示了整个地球地图的完全缩放。

MapTypeId（地图的初始类型）

mapTypeId 属性指定了地图的初始类型。

mapTypeId包括如下四种类型：

- google.maps.MapTypeId.HYBRID：显示卫星图像的主要街道透明层
- google.maps.MapTypeId.ROADMAP：显示普通的街道地图
- google.maps.MapTypeId.SATELLITE：显示卫星图像
- google.maps.MapTypeId.TERRAIN：显示带有自然特征（如地形和植被）的地图

在哪里显示 Google 地图

通常 Google 地图使用于 <div> 元素中：

```
<div id="googleMap" style="width:500px;height:380px;"></div>
```

注意： 地图将以div中设置的大小来显示地图的大小，所以我们可以在 <div> 元素中设置地图的大小。

创建一个 Map 对象

```
var map=new google.maps.Map(document.getElementById("googleMap")
,mapProp);
```

以上代码使用参数(mapProp)在<div> 元素 (id为googleMap) 创建了一个新的地图。

提示：如果想在页面中创建多个地图，你只需要添加新的地图对象即可。

以下实例定义了四个地图实例 (四个地图使用了不同的地图类型):

实例

```
var map = new google.maps.Map(document.getElementById("googleMap"),mapProp);
var map2 = new google.maps.Map(document.getElementById("googleMap2"),mapProp2);
var map3 = new google.maps.Map(document.getElementById("googleMap3"),mapProp3);
var map4 = new google.maps.Map(document.getElementById("googleMap4"),mapProp4);
```

加载地图

窗口载入后通过执行 initialize() 函数来初始化 Map 对象，这样可以确保在页面完全载入后再加载 Google 地图：

```
google.maps.event.addDomListener(window, 'load', initialize);
```

Google 地图叠加层

在Google地图中添加一个标记

Google 地图 - 叠加层

叠加层是地图上绑定到经度/纬度坐标的对象，会随您拖动或缩放地图而移动。

Google 地图 API 有如下几种叠加层：

- 地图上的点使用标记来显示，通常显示自定义图标。标记是 GMarker 类型的对象，并且可以利用 GIcon 类型的对象来自定义图标。
- 地图上的线使用折线（表示点的集合）来显示。线是类型为 GPolyline 的对象。
- 地图上的区域显示为多边形（如果是任意形状的区域）或底面叠加层（如果是矩形区域）。多边形类似于闭合的折线，因此可以是任何形状。地面叠加层通常用于地图上与图块有直接或间接关联的区域。
- 地图本身使用图块叠加层显示。如果您有自己的系列图块，可以使用 GTileLayerOverlay 类来改变地图上已有的图块，甚至可以使用 GMapType 来创建您自己的地图类型。
- 信息窗口也是一种特殊的叠加层。但是请注意，信息窗口会自动添加到地图中，并且地图只能添加一个类型为 GInfoWindow 的对象。

Google 地图 - 添加标记

记标识地图上的点。默认情况下，它们使用 G_DEFAULT_ICON（您也可以指定自定义图标）。GMarker 构造函数将 GLatLng 和 GMarkerOptions（可选）对象作为参数。

标记设计为可交互。例如，默认情况下它们接收 "click" 事件，常用于在事件侦听器中打开信息窗口。

通过 setMap() 方法在地图上添加标记:

实例

```
var marker=new google.maps.Marker({
  position:myCenter,
  });

marker.setMap(map);
```

Google 地图 - 可拖动的标记

以下实例将减少如何使用 animation 属性来拖动标记:

实例

```
marker=new google.maps.Marker({
  position:myCenter,
  animation:google.maps.Animation.BOUNCE
  });

marker.setMap(map);
```

Google 地图 - 图标

标记可以用自定义的新图标来显示，以替代默认图标:

实例

```
var marker=new google.maps.Marker({
  position:myCenter,
  icon:'pinkball.png'
  });

marker.setMap(map);
```

Google 地图 - 折线

GPolyline 对象可在地图上创建线性叠加层。GPolyline 包括一系列点，并创建一系列有序连接这些点的线段。

折线支持以下属性：

- path - 指定了多个直线的纬度/经度坐标
- strokeColor - 指定直线的十六进制颜色值(格式: "#FFFFFF")
- strokeOpacity - 指定直线的透明度(该值为 0.0 到 1.0)
- strokeWeight - 定义线的宽度,以像素为单位。
- editable - 定义用户是否可编辑直线(true/false)

实例

```
var myTrip = [stavanger,amsterdam,london];
var flightPath = new google.maps.Polyline({
  path:myTrip,
  strokeColor:"#0000FF",
  strokeOpacity:0.8,
  strokeWeight:2
});
```

Google 地图 - 多边形

GPolygon 对象类似于 GPolyline 对象，因为它们都包括一系列有序的点。但是，多边形不像折线一样有两个端点，而是设计为定义形成闭环的区域。

和折线一样，您可以自定义多边形边（线）的颜色、粗细和透明度，以及封闭的填充区域的颜色和透明度。颜色应是十六进制数字 HTML 样式。

多边形支持以下属性：

- path - 指定多个直线纬度的坐标 (第一个和最后一个坐标是相等的)
- strokeColor - 指定直线的十六进制颜色值(格式: "#FFFFFF")
- strokeOpacity -指定直线的透明度(该值为 0.0 到 1.0)
- strokeWeight - 定义线的宽度,以像素为单位。
- fillColor - 指定闭合区域的十六进制颜色值 (格式: "#FFFFFF")
- fillOpacity - 指定填充颜色的透明度 (该值为 0.0 到 1.0)
- editable - 定义用户是否可编辑直线(true/false)

实例

```
var myTrip = [stavanger,amsterdam,london,stavanger];
var flightPath = new google.maps.Polygon({
  path:myTrip,
  strokeColor:"#0000FF",
  strokeOpacity:0.8,
  strokeWeight:2,
  fillColor:"#0000FF",
  fillOpacity:0.4
});
```

Google 地图 - 圆

圆支持以下属性：

- center - 指定圆的中心点参数 google.maps.LatLng
- radius - 指定圆的半径，以米为单位
- strokeColor - 指定弧线的十六进制颜色值(格式: "#FFFFFF")
- strokeOpacity - 指定弧线的透明度(该值为 0.0 到 1.0)

- strokeWeight -定义线的宽度,以像素为单位。
- fillColor - 指定圆的十六进制颜色值填充值 (格式: "#FFFFFF")
- fillOpacity - 指定填充颜色的透明度 (该值为 0.0 到 1.0)
- 定义用户是否可编辑直线(true/false)

实例

```
var myCity = new google.maps.Circle({
  center:amsterdam,
  radius:20000,
  strokeColor:"#0000FF",
  strokeOpacity:0.8,
  strokeWeight:2,
  fillColor:"#0000FF",
  fillOpacity:0.4
});
```

Google 地图 - 信息窗口

在一个标记上显示一个文本信息窗口：

实例

```
var infowindow = new google.maps.InfoWindow({
  content:"Hello World!"
  });

infowindow.open(map,marker);
```

Google 地图 - 叠加层参考手册

Google Maps API 参考手册.

Google 地图事件

点击标记缩放地图 - 绑定在google地图上的事件。

点击标记缩放地图

我们仍然使用上一遍文章使用的英国伦敦的地图。

点用户点击标记时实现缩放地图的功能(点击标记时绑定地图缩放事件)。

代码如下：

实例

```
// Zoom to 9 when clicking on marker
google.maps.event.addListener(marker,'click',function() {
  map.setZoom(9);
  map.setCenter(marker.getPosition());
  });
```

使用 addListener() 事件处理程序来注册事件的监听。该方法使用一个对象，一个事件来监听，当指定的事件发生时 函数将被调用。

重置标记

我们通过给地图添加事件处理程序来改变 'center' 属性，以下代码使用 center_changed 事件在3秒后标记移会中心点:

实例

```
google.maps.event.addListener(map,'center_changed',function() {
  window.setTimeout(function() {
    map.panTo(marker.getPosition());
  },3000);
});
```

点击标记时打开信息窗口。

点击标记在信息窗口显示一些文本信息：

实例

```
var infowindow = new google.maps.InfoWindow({
  content:"Hello World!"
  });

google.maps.event.addListener(marker, 'click', function() {
  infowindow.open(map,marker);
  });
```

设置标记及打开每个标记的信息窗口

当用户点击地图时执行一个窗口

用户点击地图某个位置时使用 placeMarker() 函数在指定位置上放置一个标记，并弹出信息窗口：

实例

```
google.maps.event.addListener(map, 'click', function(event) {
  placeMarker(event.latLng);
  });

function placeMarker(location) {
  var marker = new google.maps.Marker({
    position: location,
    map: map,
  });
  var infowindow = new google.maps.InfoWindow({
    content: 'Latitude: ' + location.lat() +
    '<br>Longitude: ' + location.lng()
  });
  infowindow.open(map,marker);
}
```

Google 地图 - 事件参考手册

Google Maps API 参考手册。

Google 地图控件集

一个Google 地图 - 默认控件集设置：

Google 地图 - 默认控件集设置：

当使用一个标准的google地图，它的默认设置如下：

- .Zoom-显示一个滑动条来控制map的Zoom级别
- .PPan-地图上显示的是一个平底碗样的控件，点击4个角平移地图
- .MapType-允许用户在map types(roadmap 和 satallite)之间切换
- .StreetView-显示为一个街景小人图标，可拖拽到地图上某个点来打开街景

Google 地图 - 更多控件集

除了以上默认控件集,Google还有：

- .Scale-显示地图比例尺
- .Rotate-显示一个小的圆周图标，可以转动地图
- .verview Map-从一个广域的视角俯视地图

创建地图时你可以通过设置选项指定哪些控件集显示或者通过调用 setOptions() 来改变地图的设置选项。

Google 地图 - 关闭默认控件集

你可能希望能够关闭默认的控件集。

为了关闭默认控件集,设置地图的disableDefaultUI的属性为true:

实例

```
disableDefaultUI:true
```

Google 地图 - 开所有控件集

一些控件集默认显示在地图上，而其它的不会，除非你设置它们。

设置控件为true使其可见-设置控件为false则隐藏它。

以下实例开启所有的控件：

实例

```
panControl:true,
zoomControl:true,
mapTypeControl:true,
scaleControl:true,
streetViewControl:true,
overviewMapControl:true,
rotateControl:true
```

Google 地图 - 修改控件集

某些地图控件是可配置的。通过制定控件选项域改变控件集。

F举个例子来说，修改Zoom 控件的选项在zoomControlOptions指定。zoomControlOptions包含如下3种选项：

- .google.maps.ZoomControlStyle.SMALL-显示最小化zoom 控件
- .google.maps.ZoomControlStyle.LARGE-显示标准zoom滑动控件
- .google.maps.ZoomControlStyle.DEFAULT-基于设备和地图大小，选择最合适的控件

实例

```
zoomControl:true,
zoomControlOptions:  {
  style:google.maps.ZoomControlStyle.SMALL
}
```

注意： 如果需要修改一个控件，首先开启控件(设置其为true)。

另一个控件为 MapType 控件。

MapType 控件可显示为以下 style 选项之一：

- google.maps.MapTypeControlStyle.HORIZONTAL_BAR，用于在水平栏中将一组控件显示为如 Google Maps 中所示按钮。
- google.maps.MapTypeControlStyle.DROPDOWN_MENU，用于显示单个按钮控件，以便您从下拉菜单中选择地图类型。
- google.maps.MapTypeControlStyle.DEFAULT，用于显示"默认"的行为，该行为取决于屏幕尺寸，并且可能会在 API 以后的版本中有所变化。

实例

```
mapTypeControl:true,
mapTypeControlOptions: {
  style:google.maps.MapTypeControlStyle.DROPDOWN_MENU
}
```

同样你可以使用ControlPosition属性指定控件的位置:

实例

```
mapTypeControl:true,
mapTypeControlOptions: {
  style:google.maps.MapTypeControlStyle.DROPDOWN_MENU,
  position:google.maps.ControlPosition.TOP_CENTER
}
```

Google 地图 - 自定义控件集

创建一个返回伦敦自定义控件，用于点击事件： (如果地图被拖拽):

实例

```
controlDiv.style.padding = '5px';
var controlUI = document.createElement('div');
controlUI.style.backgroundColor = 'yellow';
controlUI.style.border='1px solid';
controlUI.style.cursor = 'pointer';
controlUI.style.textAlign = 'center';
controlUI.title = 'Set map to London';
controlDiv.appendChild(controlUI);
var controlText = document.createElement('div');
controlText.style.fontFamily='Arial,sans-serif';
controlText.style.fontSize='12px';
controlText.style.paddingLeft = '4px';
controlText.style.paddingRight = '4px';
controlText.innerHTML = '<b>Home<b>'
controlUI.appendChild(controlText);
```

Google 地图 - 控件集参考手册

Google Maps API 参考手册.

Google 地图类型

HYBRID类型的google地图:

Google 地图- 基本地图类型

Google Maps API 中提供了以下地图类型：

- `MapTypeId.ROADMAP`，用于显示默认的道路地图视图
- `MapTypeId.SATELLITE`，用于显示 Google 地球卫星图片
- `MapTypeId.HYBRID`，用于同时显示普通视图和卫星视图
- `MapTypeId.TERRAIN`，用于根据地形信息显示实际地图。

要通过 Map 修改正在使用的地图类型，您可以为其设置 mapTypeId 属性:

```
var mapProp = {
  center:new google.maps.LatLng(51.508742,-0.120850),
  zoom:7,
  mapTypeId: google.maps.MapTypeId.HYBRID
};
```

或者动态修改 mapTypeId：

```
map.setMapTypeId(google.maps.MapTypeId.HYBRID);
```

Google 地图- 45° 图像

Google Maps API 针对特定位置支持特殊的 45° 图像。

此类高分辨率图像可提供朝向各基本方向（东南西北）的透视视图。对于支持的地图类型，这些图片还可提供更高的缩放级别。

现有的 google.maps.MapTypeId.SATELLITE 和 google.maps.MapTypeId.HYBRID 地图类型支持高缩放级别的 45° 透视图像（如果有的话）。如果您放大的位置拥有此类图像，那么这些地图类型将会自动通过以下方式更改其视图：

- 地图上现有的任何平移控件都将会变更为在现有的导航控件周围添加一个罗盘转轮。您可以通过该罗盘来更改任意 45° 图像的方向，方法是：拖动该罗盘转轮，然后将方向对准包含图像的最近支持方向。
- 一个旋转控件将会间隙显示在现有的平移和缩放控件之间，它可用于将图像围绕支持方向旋转。旋转控件仅支持顺时针方向旋转。
- 以当前位置为中心的 45° 透视图像将会替代卫星图像或混合图像。默认情况下，此类视

图会朝向北方。如果您缩小地图，则地图会重新显示默认的卫星图像或混合图像。
- MapType 控件将启用子菜单切换控件，用于显示 45° 图像。

注意：缩小显示 45° 图像的地图类型将会还原所有更改，并重新构建原始地图类型。

以下示例显示了意大利威尼斯公爵宫45°视图：

实例

```
var mapProp = {
center:myCenter,
zoom:18,
mapTypeId:google.maps.MapTypeId.HYBRID
};
```

提示：Google 正在不断地为更多城市添加 45° 图像。有关最新信息，请参阅 Google 地图上的 45° 图像列表。

Google 地图 - 启用和停用 45° 图像 - setTilt(0)

您可以通过在 Map 对象上调用 setTilt(0) 来停用 45° 图像。要启用适用于支持的地图类型的 45° 透视图像，请调用 setTilt(45)。

实例

```
map.setTilt(0);
```

Google 地图 - 参考手册

Google 地图 API 参考手册。

Google 地图 API 参考手册

Google 地图API 参考手册

地图

构造函数/对象	描述
Map()	在指定的 HTML 容器中创建新的地图，该容器通常是一个DIV元素。

叠加层

构造函数/对象	描述
Marker	创建一个标记。
MarkerOptions	标记的选项。由DirectionsRenderer渲染的所有标记都将使用这些选项。
MarkerImage	A structure representing a Marker icon or shadow image
MarkerShape	Defines the marker shape to use in determination of a marker's clickable region (type and coord)
Animation	Specifies animations that can be played on a marker (bounce or drop)
InfoWindow	Creates an info window
InfoWindowOptions	Options for rendering the info window
Polyline	Creates a polyline (contains path and stroke styles)
PolylineOptions	Options for rendering the polyline
Polygon	Creates a polygon (contains path and stroke+fill styles)
PolygonOptions	Options for rendering the polygon
Rectangle	Creates a rectangle (contains bounds and stroke+fill styles)
RectangleOptions	Options for rendering the rectangle
Circle	Creates a circle (contains center+radius and stroke+fill styles)
CircleOptions	Options for rendering the circle
GroundOverlay	
GroundOverlayOptions	
OverlayView	
MapPanes	
MapCanvasProjection	

事件

构造函数/对象	描述
MapsEventListener	It has no methods and no constructor. Its instances are returned from addListener(), addDomListener() and are eventually passed back to removeListener()
event	Adds/Removes/Trigger event listeners
MouseEvent	Returned from various mouse events on the map and overlays

控件集

构造函数/对象	描述
MapTypeControlOptions	Holds options for modifying a control (position and style)
MapTypeControlStyle	Specifies what kind of map control to display (Drop-down menu or buttons)
OverviewMapControlOptions	Options for rendering of the overview map control (opened or collapsed)
PanControlOptions	Options for rendering of the pan control (position)
RotateControlOptions	Options for rendering of the rotate control (position)
ScaleControlOptions	Options for rendering of the scale control (position and style)
ScaleControlStyle	Specifies what kind of scale control to display
StreetViewControlOptions	Options for rendering of the street view pegman control (position)
ZoomControlOptions	Options for rendering of the zoom control (position and style)
ZoomControlStyle	Specifies what kind of zoom control to display (large or small)
ControlPosition	Specifies the placement of controls on the map

地图 API Map() 构造器

实例

创建一个 Google 地图:

```
var map=new google.maps.Map(document.getElementById("googleMap"),mapOpt);
```

定义和用法

Map() 构造器创建了一个新的地图并插入到指定的HTML元素中（<div> 元素)。

语法

```
new google.maps.Map(_HTMLElement_,_MapOptions_)
```

参数值

参数	描述
HTMLElement	规定要把地图放置在那个 HTML 元素中。
MapOptions	带有地图初始化变量/选项的 MapOptions 对象。

Map() 的方法

方法	返回值	描述
fitBounds(*LatLngBounds*)	None	设置要包含给定边界的视口。
getBounds()	*LatLng,LatLng*	返回当前视口的西南纬度/经度和东北纬度/经度。
getCenter()	*LatLng*	返回地图的中心的纬度/经度。
getDiv()	*Node*	返回包含地图的 DOM 对象。

getHeading()	*number*	返回航拍图像的罗盘航向（支持 SATELLITE 和 HYBRID 地图类型）。
getMapTypeId()	HYBRID ROADMAP SATELLITE TERRAIN	返回当前地图类型。
getProjection()	*Projection*	返回当前 Projection（投影）。
getStreetView()	*StreetViewPanorama*	返回绑定到地图的默认的 StreetViewPanorama。
getTilt()	*number*	返回航拍图像的入射角度数（支持 SATELLITE 和 HYBRID 地图类型）。
getZoom()	*number*	返回地图的当前缩放级别。
panBy(*xnumber,ynumber*)	None	通过以像素计的给定距离改变地图的中心。
panTo(*LatLng*)	None	改变地图的中心为给定的 LatLng。
panToBounds(*LatLngBounds*)	None	将地图平移所需的最小距离以包含给定的 LatLngBounds。
setCenter(*LatLng*)	None	
setHeading(*number*)	None	设置航拍图像的罗盘方向（以度为单位进行测量），基本方向为北方。
setMapTypeId(*MapTypeId*)	None	改变要显示的地图类型。
setOptions(*MapOptions*)	None	
setStreetView(*StreetViewPanorama*)	None	绑定一个 StreetViewPanorama 到地图上。
setTilt(*number*)	None	设置航拍图像的入射角度数（支持 SATELLITE 和 HYBRID 地图类型）。
setZoom(*number*)	None	

Map() 的属性

属性	类型	描述
controls	*Array.* `<MVCArray.<Node>>`	要附加到地图上的额外控件。
mapTypes	*MapTypeRegistry*	按字符串 ID 划分的 MapType 实例的注册表。
overlayMapTypes	*MVCArray.* `<MapType>`	要叠加的额外地图类型。

Map() 的事件

事件	参数	描述
bounds_changed	None	当可视区域范围更改时会触发此事件。
center_changed	None	当地图 center（中心）属性更改时会触发此事件。
click	*MouseEvent*	当用户点击地图（但不是点击标记或信息窗口）时会触发此事件。
dblclick	*MouseEvent*	当用户双击地图时会触发此事件。请注意，触发此事件前还会触发点击事件。
drag	None	当用户拖动地图时会反复触发此事件。
dragend	None	当用户停止拖动地图时会触发此事件。
dragstart	None	当用户开始拖动地图时会触发此事件。
heading_changed	None	当地图 heading（方向）属性更改时会触发此事件。
idle	None	当地图在平移或缩放之后变为闲置状态时会触发此事件。
maptypeid_changed	None	当 mapTypeId 属性更改时会触发此事件。
mousemove	*MouseEvent*	只要用户的鼠标在地图容器上移动，就会触发此事件。
mouseout	*MouseEvent*	当用户的鼠标从地图容器上退出时会触发此事件。
mouseover	*MouseEvent*	当用户的鼠标进入地图容器时会触发此事件。
projection_changed	None	当投影更改时会触发此事件。
resize	None	当地图（div）更改尺寸时会触发此事件。
rightclick	*MouseEvent*	当用户右击地图时会触发此事件。
tilesloaded	None	当可见图块载入完成后会触发此事件。
tilt_changed	None	当地图 tilt（倾斜）属性更改时会触发此事件。
zoom_changed	None	当地图 zoom（缩放）属性更改时会触发此事件。

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。